APRIL 1988: VOLUME 1, ISSUE 1

Canada $4.25
USA $3.50

Amiga Transactor

THE MAGAZINE FOR AMIGA PROGRAMMERS

Articles

Combatting The SCA Amiga Virus
by J.C. Bat

Inside CAOS - the Amiga's *Original* DOS
by Andy Finkel

A Programmer's Tour of "arp.library"
by Scott Ballantyne

Warrior Cycles - Amiga vs. Amiga
by Rico Mariani

Porting from Unix to Amiga
by John Toebes

BreakPoint –
The Art of Debugging
by Victor A. Wagner

Using AmigaDOS Packets
by Matt Dillon

The Structure of Font Files
by Betty Clay

Programming with Exec Lists
by Rob Peck

Tutorial: ML on the Amiga
by Jim Butterfield

The Ideal Programming Environment
by Perry Kivolowitz

A Baker's HexaDozen of Amiga Programming Traps
by Bryce Nesbitt

Columns

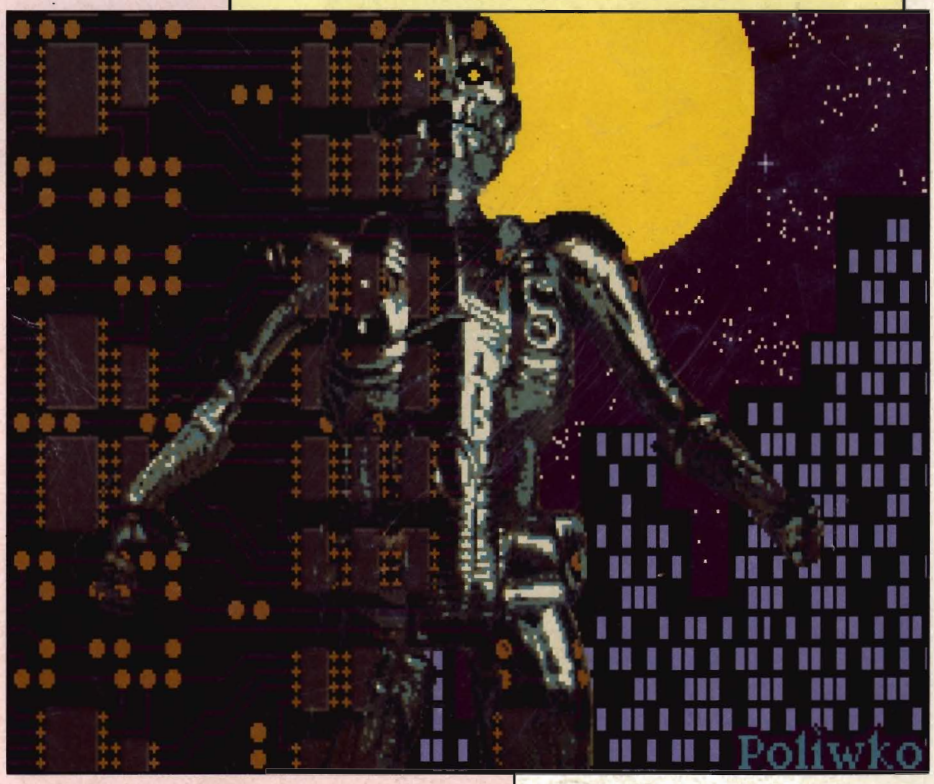
Amiga Dispatches
Don Curtis

The ViewPort
Larry Phillips

ACCESS – The best of
non-commercial software
Steve Ahlstrom

Reviews

Lattice C Compiler Version 4.0
Power Windows 2.0
saf-T-net hard disk backup utility
BYTE BOX Amiga 500 memory expansion



04
0 29074 12122 0

Congratulations *Transactor for the Amiga*, Best Wishes for a long, happy run.

THE DISK SPECIALISTS

Disk Supplies

Bulk

C.ITOH, Fujitsu

DSDD – Blue

50 –	$1.25
100 – 350	1.20

DSDD – Rainbow

50 –	$1.49
100 – 350	1.45

DSHD – High Density

50 –	$5.25
100 – 350	4.95

Bulk

SONY

DSDD – Blue

50 –	$1.29
100 – 350	1.29

M C P

Associates, Inc.
DISTRIBUTORS

featuring

C.ITOH, Fujitsu Kasei, SONY, etc:

3.5" Double Sided, Double & High Density
100% Certified – Lifetime Warranty!

Bulk – 50 disks per carton, priced per disk
Boxed – 10 " " box, priced per box.

(we honour our manufacturers' warranties)

Special Offer: AMUSE PD Disks

5 disks for $30.00

Write for listing – limited time offer. By arrangement. AMUSE – Sponsors, AmiEXPO!

Labels, Packaging

Boxed

C.ITOH, Fujitsu

DSDD – Blue

1 – 3	$15.95
4 – 9	15.45

DSDD – Rainbow*

1 – 3	$23.95
4 – 9	22.95

* includes plastic case

SONY

DSHD – High Density*

1 – 3	$55.95
4 – 9	53.95

* limited availability

Shipping/Handling Info: Minimum order: Bulk – 50 disks: Box – 2 boxes. We ship worldwide.
U.S. – Minimum Fee: $ 5.00, plus $3.50 per 100 disks bulk, or 2 boxes. For CODs, add $5.00
Canada – " " 13.00, " 5.50 " " " " " " . No CODs to Canada.
Prices subject to change and availability. Charges do not reflect any customs/duties.
MASTERCARD/VISA accepted: minimum purchase – $50.00. New York residents add sales tax.

Please note: Other brands, formats available.
Specify brand, type. Disk Supplies, Labels.
Write/Call for complete listings, and for
Quantity Disk Pricing Schedule, & Specials.

Authorized AMIGA Dealer/Education Dealer.
Systems, hardware, software, peripherals.
Corporate and Educational Accounts welcomed.
Dealer inquiries invited.

Exceptional Service and Quality

M C P Associates, Inc., P.O.Box 6260, Dept. TA, Long Island City, New York, 11106-0260

(718) 956–9000

Amiga-Tax II

Tax Planning and Preparation Software

Demo Disk Available. Send $10

Datamax Research *corp*

Box 5000, Bradford, Ontario L3Z 2A6

Pay $69.95 once then receive Yearly updates for only $29.95

- All Schedules
- All Provinces* and Territories
- PRINTS returns
- Gov't Approved
- Save time, money and aggravation.
- Use all year for tax planning
- PLUS its tax deductable!

*except Quebec Provincial

saf-T-net

The ULTIMATE Hard Disk Backup Utility!

- o Intuition-based, runs from Workbench or CLI.
- o Diskettes formatted on an as-needed basis.
- o Full or Incremental backup utilizing Archive bit.
- o Selective and full restore capability.
- o Wild-card patterns used for inclusion/exclusion.
- o Fast - approx. 3 minutes (or less) per diskette.
- o Preservation of Date & time, Comments, and status.
- o Sophisticated error recovery.

$49.95.

R.S.N. Software, Inc.
4122 S. Parker Road
Aurora, CO 80014
(303) 690-0102

Dealer Inquiries Welcome.

AMIGA OWNERS:

Discover new worlds within your Amiga™

The 64 Emulator broadens the horizons of your Amiga with access to thousands of programs written for the Commodore 64. Yes, the proven word processors, databases, spreadsheets and exciting games can now be run on your Amiga.

- Written in 100% 68000 machine code for maximum speed.
- Supports all aspects of the Commodore 64 including sound and color.
- Monochrome mode turns off the color for increased speed.
- All Amiga disk drives and printers are fully supported.
- Moving your 64 programs to Amiga disks is a snap!
- Optional Serial Interface connects any Commodore 64 disk drive and printer to your Amiga.

Why wait for an expanded range of Amiga software? Take immediate advantage of quality software already chosen by six million other computer owners.

- The 64 Emulator by ReadySoft ...here, now, and ready for you.

PHONE ORDERS (416)731-4175

YES I want to use the thousands of Commodore 64 programs on my AMIGA. Please rush me:

☐ The 64 Emulator @ $39.95 ($49.95 Cdn.)

☐ The 64 Emulator with Serial Interface @ $59.95 ($79.95 Cdn.)
(Please specify computer below)

I own ☐ Amiga 500 ☐ Amiga 1000 ☐ Amiga 2000

Payment by ☐ Check ☐ Money Order ☐ Visa ☐ Mastercard

card no. exp. signature

ReadySoft Inc.

Please add $3.00 ($4 Cdn.) for shipping and handling. Ontario residents please add 7% P.S.T. No C.O.D. please.

Name_____

Address_____

City, Town_____ Zip_____ State_____

To:
ReadySoft Inc.
P.O. Box 1222
Lewiston, N.Y.
14092

Commodore is a registered trademark of Commodore Electronics Limited-Amiga is a registered trademark of Commodore-Amiga, Inc.

AMIGA for the Transactor
The Magazine for Amiga Programmers

Editor-in-Chief
Karl J. H. Hildon

Publisher
Richard Evers

Editors
Chris Zamara
Nick Sullivan

Editorial Assistant
Moya Drummond

Customer Service
Jennifer Reddy

Contributing Writers

Steve Ahlstrom
Scott Ballantyne
J.C. Bat
Jim Butterfield
Betty Clay
Don Curtis
Matt Dillon
Andy Finkel
Perry Kivolowitz
Rico Mariani
Bryce Nesbitt
Rob Peck
Larry Phillips
John Toebes
Victor A. Wagner

Cover Artist
Nick Poliwko

AMIGA for the Transactor
The Magazine for Amiga Programmers

Volume 1, Issue 1

Thoughts On An Amiga Programming Environment 12
by Perry Kivolowitz
A probing examination of the elements that combine to simplify Perry's life as a programmer: the system hardware, software tools, physical environment. . . and pizza.

The Packet Interface to AmigaDOS in C 15
by Matt Dillon
When the high-level approach to DOS just doesn't cut it, Packets might give you the power you need. Matt gets you started with two simple programming examples.

The SCA Amiga Virus 18
by J.C. Bat
A look at the virus that's causing problems for Amiga owners: where it came from, how it spreads, what it does, how to detect it, and how to deal with it.

BreakPoint 20
by Victor A. Wagner
Vic's programming experiences from the 1960s show the early evolution of the art of debugging, back in computing's dim past. First in a series.

$13 Common Amiga Programmer's Errors . . . and how to avoid them 24
by Bryce Nesbitt
A look at common problems with current software and how to avoid making the same blunders in your own code. With examples in C.

Porting from Unix to Amiga 27
by John Toebes
A step-by-step port of a text-oriented calculator program to the Amiga, adding Gadgets, Menus and an attractive display. New I/O routines replace the old ones, but the original code is barely changed!

Entering New Dimensions An Introduction to Memory Allocation and Lists 36
by Rob Peck
An introductory look at storing your data in dynamically-allocated lists using the list-handling features of the Exec Library. A simple example program in C illustrates the concepts covered in the article.

In the Beginning was CAOS 41
by Andy Finkel
Andy Finkel, software manager at Commodore-Amiga, shares some inside information about the specifications and design philosophy of CAOS – the Amiga's ***first*** operating system.

A Programmer's Tour of "arp.library" 44
by Scott Ballantyne
ARP is not just the cry of the pelican, it's also the AmigaDOS Replacement Programs... and the freely-distributable support library that goes with it can take a lot of the headaches out of programming under DOS.

Amiga Machine Language – A Detailed Example 55
by Jim Butterfield
Jim applies his famous tutorial style to an explanation of assembler programming on the Amiga, developing an enhanced "Echo" command piece-by-piece as an example.

. . . And What Are Little Fonts Made Of? 59
by Betty Clay
Betty didn't find any puppy-dogs' tails in those font files, but she sure found a lot of other stuff. The article that will answer ***all*** your questions about font file structure.

Warrior Cycles 63
by Rico Mariani
Multitasking lets the Amiga do a lot of things with ease that would be much more difficult on conventional micros. . . such as play games with itself. Programming a Warrior Cycle is easy - how will ***your*** player fare?

Departments and Columns

AbsExecBase. . . the TransAmi editorial page 4

Amiga Dispatches 5
by Don Curtis
Don opens up his new Amiga 2000 and plugs in all the goodies, addresses complaints against the Amiga 500's power supply, and notes a variety of new Amiga products.

ACCESS – The best of non-commercial software 8
by Steve Ahlstrom
In this instalment, Steve introduces "GfxTerm" – an animated terminal program that lets you see and hear the people you're in on-line conference with – and looks at the utilities "Help" and "Where".

The ViewPort 10
by Larry Phillips
Languages and You: What language is best for a given application? Larry gives some uncharacteristically moderate answers – then lets loose and rakes C over the coals.

Reviews

Lattice C Compiler Version 4.0 73
New features and faster code make this release a strong contender in the Amiga C compiler wars.

Power Windows 2.0 77
A source-code generator that takes the pain out of creating Windows, Menus, Gadgets and more.

saf-T-net 78
A hard drive backup utility that cuts disk-swapping to a minimum.

BYTE BOX for the Amiga 500 79
BYTE-by-BYTE's new RAM expansion box lets users add their own RAM chips.

About the cover: *Synapsed*, by Nick Poliwko, Toronto.
This graphic started with an image of a toy robot, digitized with Digi-View. The next step involved taking the digitized image into Butcher, which is an image manipulation program. The palette was then re-arranged, and colours were merged in order to free a portion of the 32-color palette. The graphic was then transported to DeluxePaint II, where the circuit board design was created, made into a brush and used as a pattern fill for the left side of the graphic, as well as the black areas on the robot. All the colours were then stencilled, except for the background colour remaining on the right side of the robot. The moon was then added, as was the cityscape. The windows in the buildings were created by using a brush pattern-fill. A further modification of the palette produced the final version.

Please Note: TransAmi's phone number is: (416) 764-5273 Mondays, Wednesdays and Fridays ONLY

CompuServe Accounts
Contact us anytime on GO CBMPRG, GO CBMCOM, or EasyPlex at:

Karl J.H. Hildon	76703,4242
Chris Zamara	76703,4245
Nick Sullivan	76703,4353

Transactor for the Amiga is published bi-monthly by Transactor Publishing Inc., 85-10 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. **ISSN# 0838-0163.** Canadian Second Class mail permit pending, Gateway-Mississauga, Ont. US Second Class mail permit pending at Buffalo, NY. USPS Postmasters: send address changes to Transactor, P.O. Box 338, Station C, Buffalo, NY, 14209.

Transactor Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.

Subscriptions:

Canada $19 Cdn.
U.S.A. $15 US.
All other $21 US.
Air Mail (Overseas only) $40 US.

Send all subscriptions to: Transactor Publishing Inc., Subscriptions Department, 85-10 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7, 416 764 5273. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to our Richmond Hill HQ. For best results, use the postage paid card at the center of the magazine.

Quantity Orders: In Canada: Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario, L4K 2W7, (416) 738-1700. In the U.S.A.: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928, ask for Dave Bruescher.

Editorial contributions are always welcome. Only original, previously unpublished material will be considered. Program listings and articles, including BITS submissions, of more than a few lines should be provided on disk. Preferred format is Amiga 3½ diskettes with ASCII text files. Manuscripts should be typewritten, double spaced, with special characters or formats clearly marked. Photos should be glossy black and white prints. Illustrations should be on white paper with black ink only.

All material accepted becomes the property of Transactor Publishing Inc., except by special arrangement. All material is copyright by Transactor Publishing Inc. Reproduction in any form without permission is in violation of applicable laws. Write to the Richmond Hill address for a writer's guide.

The opinions expressed in contributed articles are not necessarily those of Transactor Publishing Inc. Although accuracy is a major objective, Transactor Publishing Inc. cannot assume liability for errors in articles or programs. Programs listed in *Transactor for the Amiga*, and/or appearing on *Transactor for the Amiga* disks, are copyright by Transactor Publishing Inc. and may not be duplicated or distributed without permission.

Production
Attic Typesetting Ltd.

Printing
Printed in Canada by MacLean Hunter Printing

AbsExecBase

As you've probably noticed, there's a lot going on in the Amiga programming community right now. Such as:

• ARP – the AmigaDOS replacement commands, and the support library of routines known as "arp.library", has lately come out from Charlie Heath, Scott Ballantyne and numerous other contributors. Whether Commodore is going to integrate ARP into the Amiga operating system seems to be still an open question right now; in any case, it's good news for programmers and users alike.

• New C Compilers – Lattice's Version 4.0 compiler is out, and Manx 3.6 is imminent, along with their Source Level Debugger. Great products both, and if ever there was a textbook example of the benefits competition is supposed to bring, this is it.

• New M2 Compilers – The Benchmark compiler was reviewed a couple of months ago in *Transactor*, but it hasn't got around much yet owing to problems between the author, Leon Frenkel, and Oxxi Inc., who at this point may or may not still be the publisher. This would be less unfortunate in a worse program – Benchmark has a lot going for it. Then there's the new M2Amiga compiler from Switzerland. We don't have any news on it yet, but it does seem that Modula-2 is threatening to become a genuine rival to C for serious Amiga software development.

• ARexx – that's a script language, imported from the mainframe world by Bill Hawes of ConMan fame, and available now. ARexx is more than just Execute's big brother; it's a complete language with a message-passing interface to other applications. One service ARexx can provide is a powerful front-end to your programs that should reduce the time and effort it takes to get an application up and running. Review next issue, we think.

There's more too. . . the Fast File System and other operating system enhancements are coming our way as version 1.3 of the OS, which is well along in development and will appear, with luck, in the first half of 1988; Obese Agnus, with 1-meg chip addressing, which could open a lot of doors for graphics-oriented software. . . the list goes on.

And in the midst of this – our first issue. We think we picked a good time to launch a technical magazine for the Amiga, and one of the things we want to do is keep you abreast of all the activity. So in this issue you'll find reviews of the new Lattice compiler and of Power Windows 2.0, a utility that we think should bring about a big improvement in the general level of Intuition-based programming; and you'll find a detailed introduction to programming under ARP by one of its principal authors, Scott Ballantyne.

We want this magazine to be a clearing-house for new ideas and developments in the Amiga programming community. A lot of what makes the Amiga so rewarding to work with is the substantial effort that has gone into creating programs that are "well-behaved" – that multitask genially with one another and are careful in their handling of system resources. This "altruistic" programming ethic was emphasized and encouraged in the Amiga's system documentation from the very earliest release of the machine, and is a tradition that should be cherished.

But there is another type of cooperation amongst programmers – that of collaborative, or at least complementary, development. ARP, mentioned earlier, is a fine example of how well this can work. Our magazine can serve as one medium for such efforts. Large-scale projects can be explored here and receive the benefit of critical examination by a large, technically-informed readership. And the end result of those efforts can also be disseminated – as with ARP in this issue – in our pages.

That is one of our goals. Another is simply to provide authoritative and useful information on all aspects of Amiga programming. We will deal in ideas as well as technique, in concepts as well as facts. You will encounter technical details here in plenty. You will also encounter more abstract things: science, history, even philosophy.

In the development of this magazine, we have been fortunate in working with some of the most highly qualified writers and software developers in North America – if you have been around the Amiga for long, many of the names in our table of contents will be known to you. Our aim has been that all of our articles should be useful to you in some way, and we are proud to present the efforts of the authors who have made that aim achievable.

We hope *Transactor for the Amiga* makes your experience of programming the Amiga easier and more enjoyable. And we look forward to hearing your comments on what we have already done, and suggestions as to what we might do in the future.

The Editors

P.S. Please don't suggest a Letters or a Bits section like their popular counterparts in Transactor Classic. Not because we don't want to do it – five pages of ReplyMsgs and BITS will appear next issue simply because there wasn't room in this one! Also rescheduled for next issue are great articles by Bob Rakosky (author of the saf-T-net hard disk backup utility reviewed on page 77), and Chris Gray (author of the Draco language compiler). Apologies to both of those guys – we suddenly found ourselves with 100 pages of copy for an 80 page magazine, and they got the short straws.

Amiga Dispatches

by Don Curtis

With this edition of Amiga Dispatches, Don Curtis takes over the column from Tim Grantham. Don is a police officer in Denver, Colorado, who has for the last two years been assigned to program design and development along with system design and maintenance of 10 AT&T Unix™ 3B2 computers. In his spare time, Don is an assistant sysop on CompuServe's AmigaForum.

I've been 'playing' with computers off and on since the early sixties. I started on an IBM mainframe when I was in college and graduated to a time-sharing system in the late sixties when I worked for General Electric. After a hiatus of about 10 years, I got involved in home computing in 1980 when I bought my first computer. I bought a computer with this neat graphics chip set designed by a guy named Jay Miner. That was an Atari 800 that I had packed full with a whole 48K of expensive RAM. I wanted to learn more about how the darn thing worked, so I started to program it. First with BASIC and then on to some 6502 assembly and a language called Action! that was a cross between Pascal and C.

In mid-1984, I started to hear rumours of a new machine with special graphic chips designed by the same Jay Miner. It was called the Amiga Lorraine. From the descriptions, I thought it was the perfect machine. I got a chance to see an Amiga in person in June of '85 and fell in love with it. I immediately signed up as a developer and got an A1000. And now, 2 years later. . . I've moved up to an A2000 with all the goodies. Speaking of which. . .

The A2000 – Getting Acquainted

The first thing I noted when I opened up my A2000 was the new manual, which is a welcome improvement over the one that came with the A1000. CBM did a good job on the introduction portion of the manual and the explanation of the WorkBench. They didn't do quite as well on the CLI commands, but at least they did go into some of the commands and how to use them. Admittedly, they still only scratched the surface of most of the CLI commands, but the A1000 manual hardly mentioned them at all. The hardware descriptions and schematics are great; unfortunately, they are for the original (West German) version of the machine, not the West Chester one, which is now the only version of the A2000 produced. The addendum sheets give you the new slot definitions. Also included are a block diagram and pinout of the new 'Fat Agnus'; too bad they couldn't have given us a new set of schematics also.

A major difference between the A1000 and the A2000 is the arrangement of the serial and parallel ports. Not only are the A2000's ports the opposite gender to those on the A1000 but, on the parallel port, they have moved six of the pins around, and one of those pins has +5v on it. A straight gender mender is not the answer here. On the A1000, pins 14, 15 and 16 were signal ground pins. On the A2000, pin 14 carries +5v, pin 15 has no connection and pin 16 is the INIT or RESET line for the printer. The grounds have been moved to pins 23, 24 and 25. On the A1000, pins 23, 24 and 25 were the +5v, no connection and INIT lines. Even though CBM says this is IBM compatible, many IBM type cables run pin 14 to the auto-linefeed input on the printer's Centronics port (where it is also pin 14). If you find that you are suddenly getting double-spaced text on a printout, pin 14 is probably the culprit. Unless you absolutely need it, it's a good idea to not hook it up at all.

The serial port changes aren't too bad if you are only using the "normal" set of RS232 lines for your modem. Pins 1–8 and 20 remain the same, so a gender mender will take care of your serial port if you are only using those pins. Other pins *have* changed, however. They've dropped the +/–5v lines and added a –12v line and a Ring In line. The audio lines have been moved and the clock lines have been dropped along with the buffered reset lines.

I did discover a little trick that greatly eases installation of cards into the A2000's slots. The expansion cards all have a metal plate on the back end. These plates have a small tab on the bottom that is supposed to go in front of a metal strip under the back edge of the mother board. The fit is very tight and, in many cases, the metal tab on the board will butt up against the top of the metal strip on the A2000 making it difficult – if not next to impossible – to just push the board into place. Pushing on the card too hard can cause all kinds of nasty things to happen to the mother board. Make it easy on yourself: take a pair of pliers and, where the tab narrows near the bottom, bend it out 1/16th to 1/8th of an inch towards the back before you start to insert the card. Bent out this way, it should easily fit past the metal strip under the mother board. It may get momentarily caught on the opening in the back of the A2000, but you can easily push in on the tab from the back of the computer to get it past the opening. Once you start to fill up the slots, you'll find that this is about the only way you'll get some boards installed.

A2000 Hard Disks and the A2090

Installing the Hard Disk and A2090 hard disk interface is pretty straightforward. The instruction manual isn't the best in the world, but I think just about anyone who is a bit handy can do

the installation with few or no problems. Don't forget the little trick above before you try to install the 2090 board. I also found it was easier to hook up the leads for the front panel LED before I put the board in place. Don't be surprised if your hard drive has some defective sectors marked on a label attached to the drive. Leave the top off the computer and use the INSTALL script supplied to set up the drive, PREP it and then format it. If you have defective sectors marked on the drive, you'll need to enter the numbers of the defective sectors during the installation. During the format, the drive will make a bit of noise when it comes to one of those sectors but, if you've entered it properly, the format will continue and properly map that sector out so it won't be used.

I've heard several complaints about problems with formatting SCSI drives attached to the 2090. My hard drive is an ST506 (IBM style) so I had no problems with the installation, but folks with SCSI drives report that it takes many tries to get the SCSI drives formatted. Once they eventually succeed, though, they report no further problems. I think this may be a problem with the software, not the hardware. The typical difficulty appears to be that the format will proceed for a few tracks, then suddenly quit. A retry may go a few more tracks and quit or it may go all the way and finish. In any event, be patient and keep trying. It will work eventually.

CBM has posted a new HDDISK driver on all the major networks; your dealer should also be able to supply you with a copy. This new driver fixes some of the problems using the hard drive while using an overscan screen. It seems that there just aren't enough DMA cycles to go around, and that was causing some read errors on the drives. The new software takes care of this problem. I ran some tests showing that the new software does slow the hard drive down a bit in reads but, since it sped up the writes by about the same amount, the overall effect is no real change. To install the new HDDISK drive, all you need to do is replace the file in your Expansion directory, yet there is no need to reformat or otherwise change your drive setup.

Janus and the BridgeBoard

The next piece to go into my A2000 was the A2088 BridgeBoard. MS-DOS isn't my idea of an ideal system but, since I use it at work, I also need it at home. The BridgeBoard is the ideal solution for me. The manual that comes with the 2088 is pretty good. Its step by step instructions and the automatic installation script file took care of the initial setup.

The manual falls down a bit in documenting the use of some of the utilities. It doesn't give any real information on using the PCPrefs program and, as you'll see later, I needed that information. The manual also doesn't have much to say about which addresses on the PC side are protected and which are available for expansion.

It's a lot easier to install the 5.25" drive if you remove the plate holding the 3.5" drives and hard drive first. This is easily done by removing four screws and pulling a couple of cables. Following my usual procedure, I gave the 2088 board itself a quick visual inspection before inserting it. CBM's quality assurance must be slipping – the BIOS ROM on the board was improperly inserted. One of the ROM's pins was completely out of the socket. . . not connected to anything. I seriously doubt that the board could have passed any kind of final inspection that involved actually running the board. I pulled the ROM, straightened the pin and reinserted the ROM. I put everything in its place, fired it up, and it worked just fine. It delivers full PC/XT speed, according to my tests, far superior to Transformer's very slow 25 to 30 percent of PC/XT speed.

The final piece to go into the A2000 was a multifunction plus Expanded Memory Specification (EMS) memory board (Emulex Persyst SB-III Fastcard IV) for the PC side of things. It seems that CBM aren't the only ones with quality control problems – the ROM chip on the board was installed upside-down. In all fairness to Emulex, the box had been opened by someone at the dealer's, and it's likely that someone had played with it before I bought it. Installation of the Emulex board brought up one of those 'Catch 22' situations. The board's ROM could be set to address any of the ROM mountpoints on a PC; however, the EMS RAM defaulted to load at E000, which happens to be where CBM installs the Janus handler. I could modify the EMS load location, but only after the machine booted. However, I couldn't boot without moving Janus! The 2088 manual wasn't much help but, as a last measure, I tried PCPrefs and, sure enough, I could move Janus.

Now the board would boot, giving me 640K main RAM and 128K EMS RAM. The board can go to 2 Meg total RAM, but I only installed 256K to start. 128K of that filled out the main RAM from 512K to 640K and the rest went to Expanded Memory. Everything looked good until I tried using JLink, ARead or AWrite. These are the shared utilities that allow the PC side and the Amiga side to share files back and forth. I had properly started PCDisk before trying to use the shared utilities, but still couldn't access the Amiga side of the A2000. It seems that the shared utilities use Janus and expect to find it located at E000. Well, now that I had reset the Emulex to load the EMS RAM at a different location (A000), I could go back to PCPrefs and move Janus back to E000.

It worked. I now have a 640K PC clone in my A2000 complete with 1 serial port, 2 parallel ports, 2 game ports, auto load battery clock/calendar and 128K of expanded RAM above 640K. For those who need PC compatibility, this is not a bad way to go. True, it's 'only' PC speed (4.77 Mhz) but is truly compatible. I've yet to find something that won't run on it.

ARead, AWrite and JLink

There is an undocumented switch for ARead and AWrite, and there are some major problems with those two when used with JLink. If you wish to move a binary file via ARead or AWrite, you must use the switch **/b** at the end of the command line. This specifies that the file to be moved is binary. If you don't use the switch, the file transfer will stop at the first ^Z (0x1A) encountered, since MS-DOS uses this character as an End of File marker.

One of the first problems you might encounter is that it is not safe to use ARead or AWrite while you have a JLinked virtual

drive. If you do, you'll corrupt some, if not all of the files on the JLinked drive. There is no documentation on this bug, nor any warning about using JLink and either ARead or AWrite at the same time, but I've encountered it every time, as has everyone else I know who has tried it. If you want to move a file from the Amiga side to the PC file via those utilities, you'll first have to unlink the JLinked drive. Use an intermediary file on a PC floppy if the source or final destination is a file in a JLinked drive.

The next problem is with JLink itself. When you create a JLink file on an AmigaDOS device such as a hard drive, what you are really doing is creating a normal AmigaDOS file that is laid out as if it were on an MS-DOS disk. MS-DOS sees this file as if it were on an MS-DOS drive complete with FAT tables, Directory sectors and the whole nine yards. Unfortunately, when you create the drive using JLink, all you create are those tables, no other filespace. The filespace comes about as you fill up the drive. As you create files on the JLinked drive, the AmigaDOS file starts to grow; however, as you delete MS-DOS files, the AmigaDOS file doesn't shrink. Once you use some space on the drive, it stays in the AmigaDOS file no matter what you do with the MS-DOS files. This wouldn't normally be a problem except that, when you JLink the file, AmigaDOS treats it as an open file. That means that if you don't close it properly, you have an invalid AmigaDOS file and lose any new data that has been added to the file. This situation only occurs if two things happen. First, you must add files to the MS-DOS virtual drive such that you cause the AmigaDOS file to grow in size; second, you have to turn the Amiga off, or warm-start the Amiga before you unlink the virtual drive. If those two things occur, you'll lose any information on the virtual drive that caused the AmigaDOS file to grow.

There is an apparent fix for this problem. The obvious fix is to not shut the Amiga down before you unlink the virtual drive. That's not always possible if you somehow or other lock up either the PC or the Amiga side of the A2000. The real fix seems to be to fill up the virtual drive when you create it. Just create several directories, and copy files into those directories until the drive fills up. This will cause the AmigaDOS file to be the maximum size you specified when you created the virtual drive via JLink. Then unlink the drive to close the AmigaDOS file and make the Amiga side happy. Once you've done that, you can go ahead and re-link the virtual drive and delete all those temporary files. You'll then have a clean virtual drive that will properly withstand shutdowns that occur before you unlink the virtual drive.

I tested this a few times by creating a directory and filling it with files so that the AmigaDOS file grew in size. I then gave the Amiga the good old three-fingered salute (Ctl-Amiga-Amiga). Yup, sure enough, I lost those files that caused the AmigaDOS file to grow. I deleted what was left of those files and then refilled the directory, again causing the AmigaDOS file to grow. This time I unlinked the drive before I did a warm-start. When I got things back up and running, the MS-DOS files were still there. Next, I deleted those files and wrote a new group of files, but kept them within the current size of the AmigaDOS file. Without unlinking, I did another warm-start. The files survived just fine. Finally, I filled the virtual drive to its maximum size, unlinked to close the AmigaDOS file, and re-linked and deleted the temporary files. Since then, I've had no more lost files in the virtual drive.

The reason this seems to work is that,. . . once you create and fill up the AmigaDOS file and then properly close it, AmigaDOS doesn't care what you do inside the file. You can change the data within the file (which is what MS-DOS does when it creates or deletes a file within the virtual drive), but that doesn't change any of the links to the file or the way AmigaDOS looks at the file, so leaving it open when you shut down doesn't cause any problems.

The Amiga 500 – Underpowered?

All reports indicate that the A500 is selling like hotcakes. Though I don't personally like the all-in-one design where the keyboard and CPU are all in one box, I'm probably not a typical user. It's a nice machine, and there are lots of folks who like that style of computer. The design was chosen to keep costs down, since the A500 had to be inexpensive (as opposed to cheap). Another consequence of that design constraint is that the power supply is suited for running the A500 itself and not much else. You can safely add the 512K Ram expansion to bring the A500 up to 1 Meg and 1 external drive, but beyond that you are really pushing it. This means that all expansion devices for the A500 should have their own power supply.

Some manufacturers have tried to blame CBM for problems with expansion devices, claiming that CBM failed to provide an adequate power supply. I disagree. CBM provided the A500 with a power supply adequate for the machine. As with motor vehicles, the manufacturer provides the car with an engine adequate for its intended purpose. If you want to pull a trailer, or otherwise heavily load down the vehicle, you have a few options: buy a car with a bigger engine in the first place, replace the engine with a bigger one, accept lower performance and possible damage to the vehicle, or don't pull trailers. With Amigas, those who need extra computing power have parallel choices: buy an A2000, replace the power supply with a heftier model, accept possible damage to the present power supply and/or computer, or don't expand with devices that draw on the A500 for power. It's as simple as that.

Amigabits

Microway is about to introduce a de-lacer for the A2000. While the input to Microway's device will be NTSC standard interlaced video, it will process the video and output non-interlaced 640×400 video. The cost is expected to be in the $600 range and you'll probably need a multi-sync monitor. . . Byte by Byte has started shipping its Byte Box RAM upgrade for the A500 (see review in this issue – NS). It comes with 0K to 2 Meg of RAM, is self-powered, and costs between $300 and $700 depending on how much RAM comes with it. . . CBM is still working on their 68020/68881/68851 (CPU, Math chip, MMU) coprocessor board. They hope to have it available in the near future. It should come with 2 Meg of RAM on board and may have provisions for more. I expect it to cost in the $2000 range. If the 68020 board makes it to market, Unix™ for the Amiga could follow.

ACCESS

by Steve Ahlstrom

The best of non-commercial software

As the primary Sysop on CompuServe's AmigaForum, and a veteran Amiga programmer, Steve Ahlstrom sees just about every notable public domain and shareware program that gets written, and in this column he tells you about the ones that have recently caught his eye. If you have some information you'd like to share through this column, you can leave a message to Steve on AmigaForum (his PPN is 76703,2006), or – if you're not a CIS subscriber – send the info to us and we'll pass it on!

GfxTerm

Most of the commercial information services have an area where real-time communications between users can take place. The idea is somewhat like CB radio. You are in a 'room' with many other people. Instead of talking through a microphone, you type. When you hit return, your text is sent to all those in the room with you. It sure beats trying to carry on an ongoing conversation through the message base system. And now the world of real-time Amiga telecommunications has just taken a giant step forward on CompuServe's AmigaForum!

Introducing GfxTerm (well, the name is no award winner!) by Micro Cybernetics Corporation (MCC). GfxTerm is tough to describe in words. It's something that almost has to be experienced. If you've seen ACO (another visual conferencing program available on PeopleLink) you have seen a glimmer of what can be done. When executed, GfxTerm starts out as a rather nondescript simple terminal program. Now log onto CompuServe and enter the AmigaForum's CO area. Click on "Multi Window Mode" – this is where the fun begins!

The first thing you'll notice are the faces and/or pictures that begin popping up on your screen from the other people in CO using GfxTerm and, probably, a cacophony of sound coming from your speaker. (Don't worry, though, you get used to the sound or have the option to turn it off). Visually, you'll see little blue robots, all talking, with mouths moving, sending you different expressions, maybe some arms waving 'hi' to you, as you announce your presence in CO, and probably a few rounds of applause, or a jokester or two sending you a few catcalls! Yes! You've got digitized sound that you can send and have heard by everyone else in the CO with you.

With all of the visual scenery going on, you might think it hard to keep up with what's being typed by the participants. There is a small window at the bottom of the screen that scrolls the text. If you're quick, you can read the text, but, with so much going on you may miss things. No problem – Narrator speaks for you! I find it very easy to just sit back and *listen* to what is being said instead of trying to concentrate on reading every line that comes by. If you want, press the ESC key, to bring an all-text screen to the front. There you can see anything you may have missed before. You can also have the entire session captured for offline reading. By hitting ESC again you go back to the graphics screen.

Now that's a lot happening! But, let me try to back up a bit and explain how a few things work. The first time you use the program, each person in the CO using GfxTerm will appear on your screen as little blue robot faces. There are 10 different faces that can be used and 6 different arms/hands that can accompany the face, or be displayed by themselves (3 right arms and 3 left arms). Depending on the conversation, you may be seeing surprised expressions, sad expressions, glaring expressions, and so on. Because the program allows for animation, you may be seeing waving hands, people scratching their heads, or even more complex animations (more on that later). There are 10 different digitized sounds you can send: catcalls, applause, laughter, yawns, and more. You can turn the digital sounds on or off globally. Narrator can be turned on or off either globally or for specific speakers. You can even turn off the graphics coming from specific people – total control of the environment from your end.

But that's only half of it! What I've tried to describe is a first time session using the program. The robot faces and arms are the default – you can create your own (and you're highly encouraged to do so). Just draw your faces and arms using Dpaint or any other paint program that can produce IFF brush files. You have a 16-colour palette to work with. It was chosen carefully to allow for all skin and hair types with enough different shades to do realistic shading. You are not forced to create all 10 faces and 6 arms, nor do they necessarily have to be 'faces' and 'arms'. Many people create 4 or 5 standard faces and use the others to take advantage of the animation features. You can have sparkling winks, expanding and exploding heads – almost anything you can imagine. The same applies for the arms. Arms can be positioned in front of or behind the faces. Because depth arrangement is allowed, many interesting visual effects can be created. Some 'arms' that I've seen have been business cards, or signs that say something to the effect of "I'll be right back", and alter egos sitting right next to you. You are limited only by your own imagination.

The individualized face/arm brush files are used as input by a program that creates a personality file or 'pfile', which is written to disk. You then upload your pfile to AmigaForum for others to download so they can 'see' the real you. If you don't have the pfile for someone in the CO, you will see the robot faces as a default. These pfiles are ARCed together in a master file once a month so you can download everyone's pfile at once. They are also available singly so you can download just the people you know, get the newest ones, or get updates as people change theirs. Since pfiles include user information, such as your CompuServe ID number and your name, it is essentially impossible for anyone to masquerade as someone else by misappropriating a pfile.

The program has many other features too numerous to describe. You have Function-Key support for the simple terminal portion of the program. There are 10 macros for you to store animation and sound sequences. You can customize the parameters for the Narrator in your pfile. You can choose your own voice pitch, speaking rate, male or

female voice, natural or robot voice. When you use this feature, it becomes easy to recognize who's talking strictly by the sound of the voice. You even have a preview window to try out your animations before sending them out for all to see (and to make sure you don't send a laughing face in response to a sad story).

The preview window is the real key to the animation. In this window, you can edit and preview animations and digitized sounds before you send them. Creating animation sequences is very intuitive. There are no codes to type in, no formulas to fill out. All you do is grab an arm or a face with the mouse and move it. You can set delays between the movements by clicking on the Delay button. Mixed into the movement sequences you can embed sound sequences. It's so easy it's scary! Once you've designed a sequence that you like, you can save it to a macro key so it can be saved to disk and reused later.

The authors hint that future releases of the program will include a customized body part positioner/programmer and the possibility of using personalized digitized sounds as well as the 10 default sounds.

As I'm writing this (middle of November), GfxTerm is in large scale beta testing. Since I am basing this review on a near-final release beta version, some things may change a bit. Public release of the program is expected to be the first or second week of December, so it should be quite widespread by the time you read this. It was shown privately to a selected group of people at the AmiExpo show in New York. The response was very enthusiastic.

GfxTerm is shareware. The suggested shareware price is $25. The fully functioning program, just as described, will be found in DL 4 of AmigaForum.

GfxTerm is just a small first step in MCC's plans in the telecommunications world. When asked to elaborate a bit, company representative Ariel Butler said, "We are working on a general purpose terminal program that incorporates many of GfxTerm's features, plus enhanced ability as both a mainstream terminal program and a comprehensive conferencing tool."

While I'm sure that's as vague as it was meant to be, I'm also sure we'll be seeing some exciting future products from MCC.

GfxTerm is one of those programs that really makes your Amiga shine!

Help

On most UNIX systems, there is a program called MAN(ual). It's really a very primitive database program whose basic function is allow you to get to on-disk documentation of various programs. For instance, if you are trying to compile a program and don't remember the switches you need to use, you can type "MAN cc" at the prompt, and documentation for the compiler will scroll up on your monitor.

Help is very similar in concept to MAN. You have an ASCII data file, named MANUAL, that is kept in your S: directory. When you type Help <keyword> from the CLI, the data file is sequentially searched until the keyword is found. If the keyword is found, up to a screenful of information will be displayed on your screen.

Help is really very short and to the point. Any kind of textual information can be kept in the data file. The HELP.ARC file, as distributed on various BBSes and commercial services, contains the program, documentation, and a data file. The data file consists of all of the AmigaDOS commands and descriptive text on their usage. The program is useful immediately (have you ever wanted to use a relatively obscure AmigaDOS command and had to go searching for documentation on how to use it?).

Since you can customize the data file in any way you wish, Help can become a handy reference tool. I know one person who added the names and descriptions of all of the Amiga system calls. If he doesn't remember the parameters to Text() he can type Help Text(). I use it as a phone number/address book among other things.

As mentioned above, the data file can contain any text information that you wish. It is a straight ASCII file and can be edited with practically any text editor. There is at least one support program for Help, written by John Sobernheim, called Remember. It allows you to add entries to the data file without the need of an editor. (I myself use Rick Stile's Uedit. A comprehensive look at his latest revision will be coming in the future.)

Because the program does a sequential search of the data file, once that file reaches 50k or more, you may experience a noticeable slow down in performance when the entry you are looking for is toward the end of the file. The authors (4th Works, Inc.) say that future versions will incorporate a disk indexing scheme so the program can find the entry immediately if it exists. Until that time, if you have a large data file, the only real speed up is to run it from a hard disk.

Help is copyrighted, but is not shareware and is freely distributable.

Where

I probably overuse the AmigaDOS "path" command. My startup-sequence adds 10–12 directories to the search path. For me, it's very useful. It allows me to run a program from almost anywhere without having to worry about what device or directory I'm currently logged onto. There can be a down side to the power that "path" gives you, however. Occasionally I forget just where a certain file is! When this happens, I can always type "path" with no arguments at the CLI to see what my search path currently is, then DIR each of those directories individually until I find the file – a time-consuming approach. That's when the programs Where and WhereIs come in handy.

Where <filename> will find where a file is hiding! Where starts looking for the file in the current directory and threads its way through all of its subdirectories. If the file is found, it will print out its full pathname to the CLI. So, if you are looking for a file that you think is on a certain disk, CD to the root directory and run Where, and the entire disk will be searched. Where will exit once it has found the first occurrence of the filename.

WhereIs does the same job as Where, but does not terminate after the first occurrence. It will continue searching the disk for multiple copies of the same file or multiple files with the same name.

Wrapping it up

There must be 50 or so "DU" or disk utility type programs. For many operations these utilities are a real time-saver. Because there are so many of them available and because they all have slightly different features that you may or may not need, I'll be taking a good look at several of them for next time.

by Larry Phillips

Languages and You

Larry Phillips is an Amiga hardware and software hacker from Vancouver, British Columbia, and a Sysop on CompuServe's AmigaForum.

Have you ever stopped to think what you are doing when you write a program? I don't mean just that you are telling the computer how to do a certain task. I am speaking more of what is in between your thoughts and the final result, the executable program. The compiler, assembler, or interpreter you use to translate your intentions into a sequence of instructions plays a very important part in the programming process.

So, which language do you use? Which is the best language? As with many other things in life, there is no hard and fast answer, regardless of what the proponents of any given language tell you. If you are thinking that certain languages are more suitable for a given task than others, you are right, but only partially so.

We tend to categorize languages: FORTH is good for process control, C is good for writing operating systems, Lisp is good for list processing, Prolog is for Artificial Intelligence and "expert systems", and so on. The problem with these blanket categorizations is that they tend to be self-fulfilling. A programmer will reject a language for a given project if he has categorized it in advance as being unsuitable. On the other hand, he may also try to bend his favourite language to fit the task.

Consider LOGO. Years ago I wanted to learn more about LOGO, and asked many people where I could get more information, or perhaps sign up for a course. The reaction was almost universal. "You don't need a course, just play with it – the turtle is easy." "Oh, is that the turtle graphics language?". When I finally did locate a place that gave a course in LOGO, I was disappointed to find out that it was really just a course in turtle graphics, and taught none of the powerful features of LOGO. Seymour Papert, the father of LOGO, intended the turtle commands as the "baby talk" of the language, yet for all the computer community knows about LOGO, Papert may as well have stopped with the turtle commands.

This example is one of the more blatant instances of preconceptions limiting the use of a language, but LOGO is by no means the only language to suffer this fate. Among the more popular languages, the proponents of each one will argue the merits of their favourite with what approaches religious zeal, dismissing the benefits of the others out of hand. This kind of advocacy is less often based on fact than on unexamined perceptions and preconceived notions. In any discussion of languages, you will hear things like:

"It doesn't let you get down to the (bytes, hardware)."
"It's too (verbose, terse, structured, unstructured)."
"It doesn't do (I/O, floating point, strings) well."
"It's too (strict, lax) in its type checking."

I am going to go out on a limb here, and see if someone can saw it off behind me. I am going to say that most of the above are true statements when applied to the right language. Yes, Pascal does not let you get down to the bytes. Icon does not let you get down to the hardware. C is too terse. Modula 2 is too verbose. FORTRAN doesn't do I/O well. PILOT doesn't have floating point (or at least most PILOTs don't).

Having said that, let me also say that the truth of the statements is not only dependent upon the language, but on the program being written, the programmer writing it, the implementations of the language available, and the native resources of the machine that can be used from the language. While this sounds sufficiently vague, and contains enough qualifiers to ensure the safety of the limb I have crawled out on, it deserves a closer look.

Let's look at a few programming problems. I am going to arbitrarily pick some problems that the average person might want to solve.

1. Generate an ASCII chart.
2. Control a tool, such as a milling machine.
3. Calculate the results of any expression entered.

Now, if you were to gather proponents of a number of different languages together and ask them all to write the above programs in their favourite language, the results might be interesting. In the case of generating an ASCII chart, which programmer would finish first? Which program would be smallest? Fastest? Would any of these "wins" be important? The answer, of course, depends on the use to which the program would be put. If it is to be used only one time, writing it quickly would be more important than the size of the program. Being small and fast would be more important if it was to be used often, as might be the case if it was intended as a reference available 'online'.

The next example is one that should have all FORTH and assembler programmers reaching for the saw, thinking that my limb is vulnerable. They will be saying that no other languages could possibly be as good for direct control of hardware.

The last one is likely to be used more than once, so size and speed are more important. It sounds like a simple problem on the surface, and some languages handle it easily. Other languages would require a full blown parser, input routines, error checking, and so on. I will warn you in advance that this is a carefully chosen entry, a "trick question" if you prefer.

In each of these cases, there are languages that are clearly the wrong tool for the job, either because they just can't do certain things, or because they would need too much "bending" to accomplish it.

Looking at the ASCII chart, we can see that if it is a one time program, a high level language would be better than assembler, as the problem lends itself well to a structured approach. Among high level languages, an interpreted language is probably the best choice, because of the relatively quick turnaround in the coding/testing cycle. Which high level language you choose should be based on something else. That something else is really the crux of this article. It should be based upon the way you look at the problem, vs. the languages you know and have access to. Everyone has Basic on their machine these days, and it might be a good choice. COMAL is another, as are Icon, Lisp, and even APL. It really depends on how you see the problem. If you can easily see how to do the job in Basic, but can't imagine how to go about it in APL, then Basic is obviously the right choice for you, and vice versa.

Direct control of hardware is one of those areas where a lot of controversy seems to be the norm. FORTH programers will tell you that there is no better language for the job. Assembler programmers will say the same about their pet language. This type of program is by no means limited to those languages, however. Of the many others that are suitable, C and Modula 2 come to mind immediately. Both are capable of "getting down to bare metal" with absolute addressing, both are fast enough for mechanical control applications, and offer more structure than do either assembler or FORTH. Again, we have the situation where the language of choice is really a matter of how easily we can see the problem as it relates to the language. Do you like a bottom-up approach? Top-down? Can you view the individual parts of the problem in abstract terms or do you think along more concrete lines? The answers to questions like these will determine what language you feel is best suited to the task.

I said that problem 3, calculating the result of any expression, was very carefully chosen. Consider the problem in most languages. You will need to accept input, check it for validity, then parse and evaluate the expression and print the result. This is obviously quite a complex task in assembler, FORTH, Basic, etc., while other languages have higher level constructs and built in routines to make the job easier. Input and validity checking is relatively easy in many high level languages, as is output of the results. The tough part of this program comes when we attempt to evaluate the expression. This requires some sort of parser, and if you have ever tried to wade through an explanation of the various types of parsers in a book about say, writing compilers, then you will see that it is not a trivial task.

The trick part of this comes into play because a few languages provide for evaluation of a string with a single keyword. ARexx (see note) on the Amiga is a command language, similar to, but more powerful than the script language provided with the Amiga. It is like the REXX you will find on IBM mainframes, or like Personal REXX, found on the IBM PC and its clones. Here is the complete program:

```
arg x
interpret 'say ' x
```

I told you it was carefully chosen. Contrast this with assembler, C, Modula 2, and the the others, and you will see that ARexx is clearly better for the job, provided you have ARexx and use it for a number of other programs. Among other high level languages, there are a few with a similar facility, including COMAL, LOGO, and Lisp.

What has all this been leading up to? Well, the whole point is that languages are very personal. They are close to our souls, reflecting the way we see problems and the world in general, and a criticism of our favourite is taken as personally as an attack on our deep down values. Remember this the next time you are arguing with a programmer whose preferred language is not your own. Remember that he sees things in different ways, and that those ways, unsuitable as they may be for you, are most certainly suitable for him.

If you like to program, or are considering a career in programming, you might also want to think about this: the more languages you know and can use, the more likely you are to be able to tackle a problem and do a good job on it. A carpenter cringes when he sees someone pounding in a nail with the handle of a screwdriver or driving in a screw with a hammer, and programmers are no different when it comes to using the right tools. One of the beautiful things about programming, though, is that we have many tools that are suitable for the same tasks; you can pick the tool that is most comfortable, allowing you to concentrate on the problem, instead of wrestling with the tool.

* * * * *

Please note that all of the above was written in an uncharacteristic fit of moderation. In other words, I too suffer from the same prejudices as anyone else who has ever come across an unsuitable language. I do feel very strongly about certain languages and their suitability to programming. I won't go into the 'special purpose' languages that are meant for a limited set of tasks, but will confine my few comments to the languages that are considered by many programmers as being suitable for general purpose use.

The C language, touted by its proponents as being terse, portable, efficient, extensible, flexible and, if they thought they could get away with it, the language that will grow hair and make you younger too, is not my idea of a good language.

My main beefs with C are:

- It is inconsistent in its application of the 'rules'.
- It is neither high nor low level as languages go.
- It is very difficult to read, due to the excessive use of punctuation for operators and block structuring delimiters.
- Far too much of the language was 'left up to the compiler writer'.
- Separate modules aren't separate at all, and can interact in catastrophic ways as others are changed.

The C learning curve is steep, and not one to be scaled without collecting a lot of bruises. One must always be aware of the traps awaiting the programmer. These traps are set by the compiler implementation as well as by the so-called definition of the language, a definition which consists of a rather rambling, contradictory, and incomplete specification called "The C Programming Language", by Brian Kernighan and Dennis Ritchie. Currently, the ANSI standards committee is working on producing a standard for C.

. . .cont'd on page 14

One Hacker's Thoughts On An Amiga Programming Environment

by Perry Kivolowitz

Perry Kivolowitz is co-founder and president of ASDG Inc., manufacturers of the 8MI 8-megabyte RAM expansion board for the Amiga 2000, FACC II (the floppy accelerator) and other well-known Amiga hardware and software. He has contributed to the Amiga community with the ASDG Recoverable RAM Disk (RRD). He is currently working on firmware for ASDG's upcoming Satellite Disk Processor, and a new arcade-style game called CubeMaster.

There's a branch of computer science that concerns itself with identifying the ideal programming environment. Researchers in this field, which borders on a social science, seem to think that if you can surround Joe BitDiddler with enough sophisticated tools the Midnight Hacker will soon emerge.

Be that as it may, I thought I'd share with you a description of the programming environment I've built for myself on my Amiga. A lot of what I have to say should be taken in a "your mileage may vary" frame of mind. These things work for me – that's the only warranty I offer on the advice contained herein.

By the way, two things should become obvious. First, I've expanded the term "programming environment" beyond the purely tool-oriented technical scope to which it is usually confined. And second, this will not exactly be a dry discourse. Again, take this for what you will.

Straight Through to Monday Morning

Productive hacking doesn't take place in a few minutes here and there. Long contiguous spans of hours must be committed if you expect anything useful to come of your efforts in any reasonable period of time. To avoid interruptions from the telephone I generally hack the weekend. Friday night. . . to Sunday afternoon. Straight through. Sunday afternoons are when I get to sleep late. Straight through to Monday morning.

A popular myth is that all truly inspired hacking takes place after midnight. I myself have found this to be completely false. Rather, all truly inspired hacking takes place after two a.m. There's just something magical about the hours between two and seven in the morning. Whoever originated that rubbish about midnight was obviously a lightweight.

Get yourself a gratuitously large desk, perfectly flat with just the right amount of texture to give your mouse some traction. Hacking on a small desk is like taking seven final exams with only a 3.5 inch diskette as a work surface. Hacking needs space. Reference material – like manuals (more on this later), soiled cups (more on this later too) and pizza boxes must all comfortably share your work space.

The 12 to 16 Inch Circular Debugger

Pizza is the true food of the hacker. I know there are some hackers out there who want you to believe they're from the quiche and asparagus set, but you listen to me. You put one of these avocado heads in a locked room with a persistent guru meditation for longer than a day, it ain't wheat germ that'll power them through the tough spots. Uh-uh. A 12 to 16 inch circular debugger made from grains, tomatoes and certain milk products are what's going to save the day.

On a related topic, there are those who feel that caffeine takes a prominent place in the hacker's diet. This may be true. This may be true. This may be true. What may be true? Never mind.

A comfy chair is essential to all hacking efforts. It is only a rumour that the thinking faculty resides solely in the head. Try sitting on a less than comfy fixture for 30 hours and you tell me which part of your body is doing your thinking.

Seriously, the number two occupational hazard of the computing professional is problems relating to the back and arms – problems that can be significantly reduced, if not eliminated, by proper seating. Face it, you cannot be productive if you're in pain. I strongly suggest you contact your local occupational therapy association (OTA or AOTA) for tips on what you should be looking at for proper seating. By the way, I am sitting in a flimsy metal folding chair right now. I am regretting it.

The number one hazard of the computing professional? Your eyes of course. Expect to go blind before you are 200.

By the way, health professionals such as occupational therapists can show you simple exercises that can greatly alleviate physical stress and thus greatly increase your productivity. (Jeez, I can hear Captain Crunch nodding in approval from here!) Comfort. . . is productivity.

Programming on the Amiga should definitely be done in a room in which incandescent lighting is available. Fluorescent lighting is good *most* of the time, but evenly distributed indirect incandescent lighting will be good *all* the time.

Hacking on the Amiga cannot be done from an IBM PC. I'll bet you can guess which popular Amiga packages were cross developed on an IBM PC. Think of ones that don't work with all Amiga configura-

tions or operating system versions. Cross developing for the Amiga from an IBM PC is not only insulting, it's like covering the war in the Persian Gulf from a bar in Hoboken N.J. You want to develop for the Amiga? Develop *on* the Amiga. If you don't, your product will probably show it.

Four Meg Minimum

I have found that four megabytes of memory is the minimum amount of memory needed to effectively develop on the Amiga. But then, of course, ASDG Incorporated makes memory products for the Amiga, so my view may be skewed. I really do find myself restricted with less than four meg, though.

What are you going to use that RAM for? Well, for instance:

A recoverable RAM disk is essential. I happen to know of a good one at a real good price (the author and I are quite close). A product like FaccII is also critical to producing an effective working environment.

Now don't get me wrong. I am not saying "Go Buy FaccII (and the RRD) because I wrote them and I wrote this article." I *am* saying "I spend a lot of my life developing software on the Amiga. I wrote the RRD and FaccII *because* they make the Amiga an effective working environment." Basically, I spend my life working on Amigas. FaccII and the RRD were written to make my work go faster. They'll make your work go faster too.

I put the entire C: directory into my RRD, as well as, in fact, everything else on a normal Workbench disk. My startup-sequence does the appropriate magic to enable DF0: to be replaced with *any* diskette, and at no time will the system ask for my boot disk again for any reason (except after a crash, but even this can be eliminated). This way, I get an extra floppy drive on my system.

I also put the entire C language environment in the Recoverable RAM Disk. Also, I'll put the entire source code of several projects into the RRD to be used as needed. I often work on more than one project at a time and keep them distinct by placing them in different directories.

Occasionally, if there's a RAMdisk-killing bug yet to be found in the code I am developing or, more commonly, if I am using a commercial product that contains glaring mean nasty ugly obvious bugs that kill the RAMdisk (like Deluxe Paint II, which will kill a RAMdisk if there are too many files in a drawer when you bring up the file requester), I might choose to develop with the source code right on floppy. Again, I don't want to appear too commercial here, but developing off floppies would be unthinkably slow without a program like FaccII.

The Supreme Authority

Keep plenty of tools on line. Generally I'll keep the autodocs disk in one of my drives (I use three floppy drives). I mentioned manuals before and said I'd talk more about them later. Well, let's just say that the autodocs on disk are the supreme authority. The Amiga manuals to this day are still not to be totally trusted. If in doubt – get out the autodocs.

Since I keep the entire C language environment in the RRD, the include files are also on line. Use them. They are a great source of answers.

I myself use Manx C exclusively. I must state that this has nothing to do with the current merits of Manx as opposed to Lattice. It has to do completely with the merits of Manx as opposed to Lattice almost two years ago. Both products have come a long way since then, I understand, but I must tell you that I haven't seen Lattice for almost two years, so I can't speak from first hand experience.

I will be trying to get back into Lattice in the near future to see what all the excitement is about but, as I said before, I've been using Manx exclusively for almost two years. By the way, the Amiga community is damned lucky to have the likes of Jim Goodnow (of Manx) and John Toebes (of Lattice) working with them. I wish I could get them to work for me altogether. (Anyone who knows how picky I am concerning coding will recognize from that statement the magnitude of the professional respect I have for these guys.)

Using the right editor is another paramount choice. But more so than anything else, choosing an editor is a religious issue. So much so that, even in this article, which might well be sub-titled "The Gospel According To Perry", I will avoid getting entangled.

I myself used the CBM supplied Emacs editor for a long time. But, finally, Z from Manx won out and now I use it exclusively, except when I don't use it. In which case I use Emacs. Or is it the other way around?

Free Communications Programs

The other tools I would suggest you keep on-line include: VT100 from Dave Wecker and others, or Comm from Dennis James. There are so many excellent free communications programs available for the Amiga that anyone considering doing yet another commercial package of this type should think long and hard before doing so. That is not to say that it's not a good idea. Just be sure of yourself and your product before you invest a lot and get little.

Look into NGI for converting brushes to hex arrays. There are several other public domain programs of this type available. Programs of this nature are essential for giving your offerings their own unique look.

A debugger, of course, is an important tool to have around. I use db from Manx. In the future I will try out SDB from Manx, as well as MetaScope from Metadigm, which I hear is an excellent product. I have to say, though, that the most powerful debugger ever developed – and the one most frequently underused – is the one between your ears (or between your arm rests – depends how long you've been sitting, remember?)

I could write several hundred pages on the subject of debugging (in fact I already did – I wrote a few courses on the problem-solving aspects of debugging), but suffice it to say that most people can't debug because they aren't listening to the problem.

For developing system code on the Amiga, an essential debugging tool is a remote terminal running at 9600 baud connected to your serial port. Using "debug.library" makes short work of many

seemingly complex bugs. You can use either a terminal or another Amiga as your remote debugging station.

Having a second Amiga around during development does dramatically increase your productivity. Develop on one Amiga, crash on another – you get the idea. A recoverable RAM disk makes your rebooting time minimal, but no rebooting beats all.

One thing I make it a point to do is make sure my code is tested on a 512K machine, a 68010 or 020 based machine, and a PAL (as opposed to NTSC) machine before shipping a product. Testing. Testing. Testing.

I repeated the word "testing" three times because it is apparently a new word for many Amiga developers. Don't you people test your code before you ship it? Sheesh.

Sure it's romantic to hear about "We fixed that bug or added that feature at three a.m. this morning, and now we're shipping a thousand copies an hour." That's bogus, man. In the real world you don't ship new code until you've given it a chance to build up to its true bouquet. For example, Facc and FaccII were in constant use by six people for about six weeks before they were shipped. Nothing goes into production without time to age and mature. Some software providers follow this rule, others don't. I'll bet the consumer can tell which is which.

The Dreaded P-Word

Occasionally it is really helpful to print some of your source code out for a visual inspection. To do this, of course, requires either a printer of some sort or very sharp and very tiny finger nails. Printing your source code is good because you can stretch the resulting (shall I dare say the dreaded P-Word? Sure. . .) *paper* all around you (on your gratuitously large desk, even). In addition to having a generally calming affect, seeing your code all around you allows you to grok more code at one time.

You see, we humans generally understand those things we can see all at once better than those things we can see only 24 lines at a time. You'll find your coding will improve simply by examining your printed code from time to time, or by switching into interlace mode in your editor to get more lines on the screen at once. (Of course, if you work too long in interlace mode you bump into primary hazard number one – remember?)

I don't currently have a hard disk (not that I'll tell you about here anyway), so I'll reserve statements concerning the effect of a hard disk drive on your Amiga programming environment until a future time.

I can say, though, that I used to program regularly on an unloaded VAX 11/785. At a few hundred thousand dollars it makes a swell single-user work station. For a lot less than that I've built my Amiga into a development platform as good as, if not better than the VAX.

In this trip through Wonderland I've tried to describe how I've done that. If there's interest, maybe the good folks at TransAmi will suffer through another one of these treatises on the same subject. If not, well then, so long and good hacking to all.

cont'd from page 11. . .

This effort is a noble one, and could actually produce a standard that would allow a compiler writer to know when his compiler behaves in a way that all other compilers will behave, if they too follow the standard. It may take a while, though, as there are areas of "The C Programming Language" that have been interpreted in many different ways.

C has been compared to, and indeed referred to as, a sophisticated macro assembler. To my mind, an assembler is at least honest, not pretending to be a high level language. C will hide details from you, providing the equivalent of macros for some very powerful constructs but, at the same time, hiding things that are best left in the open if we are to consider it to be a sophisticated assembler. As a high level language, it falls far short. It will tell you when you have made fatal errors in assignment between types, failed to declare something in advance, and so on, but it tells you these things only in some cases, while assuming that you know exactly what you are doing in other cases.

Take casting as an example. Modula-2 is often criticized for being too strict with its type checking. C aficionados will say that they do not want the compiler to 'hold their hand', yet they accept the danger that a cast that is automatically done by the compiler will have disastrous results. That these results may not be seen until some future date, when the value contained in a particular variable gets altered in an unexpected way does not seem to bother the C programmer. Should not a compiler either check all type coercions or check none of them? In a strongly typed language like Modula-2, the compiler does indeed check at compile time for type mismatches, and will not allow you to continue until you have corrected the problem in one of two ways. One way is to correct the error you have made in trying an incompatible assignment. The other way is to explicitly coerce the type (called 'casting' in C) to match what you are really trying to do. In either case, you are forced to think about the consequences of your actions, leaving you in control of the code instead of at the mercy of the compiler's ideas of what you meant. In this way, you are far less likely to be bitten by a hidden bug later on. It has been said that "everything is legal in C, until run time", and there is a lot of truth in that statement.

All in all, I consider Kernighan and Ritchie's clever hack to be just that, a clever hack, without much going for it in larger, real world applications. Generations of buggy code in large operating systems attest to the unsuitability of C for anything but small, one programmer projects. Side effects abound, data cannot be hidden, nor can modules be independently 'proven' if even one other module has been changed.

No doubt I will have touched a few nerves with these statements, and I would be very interested in hearing what you look for in a language. For the record, my favourite languages are assembler, COMAL, Modula-2, and Icon, not in order of preference, but depending upon what it is I am trying to do at the time. So what languages do you use? Why? What are your beefs against my favourites?

Note: ARexx is available from: William Hawes, P.O. Box 308, Maynard, MA 01754, tel. (617) 568-8695. The cost is $49.95 (US).

The Packet Interface to AmigaDOS, in C

by Matthew Dillon

Dealing with DOS at the Packet level presents some new programming options

Matt Dillon is an Electrical Engineering senior at Berkeley. He has been working on software and hardware hacks beginning with his first microcomputer – a Commodore PET. He started with the Amiga almost two years ago (his mind was set on the machine from the time he first read its technical specs), and is known in the Amiga community for his widely used DOS shell and his text editor, "DME".

The inner workings of AmigaDOS have been a mystery for a long time and, until recently, most technical discussions of it have been in the realm not of the programmer, but of the programmer-guru. I hope to break this mould by presenting an article that will not only provide a good introduction to the workings of AmigaDOS, but also some practical applications in the way of examples in C. This particular article deals with interfacing to AmigaDOS from an application's point of view, and is by no means all-encompassing.

First, some background. AmigaDOS is rather strange in that it was written in the BCPL language. Although many good things can be said for BCPL, the fact remains that it doesn't integrate well with the rest of the Amiga's operating system. Specifically, pointers, strings, and libraries are handled differently in BCPL than in C. A BCPL pointer (BPTR) is always longword aligned. In fact, it has no choice but to be longword aligned, because BCPL pointers are stored in a divide-by-4 format. The following macros can be used to convert between C pointers and BPTRs:

```
#define BTOC(bptr)   ((long)(bptr) << 2)
#define CTOB(cptr)   ((long)(cptr) >> 2)
```

BCPL strings (BSTRs) are not like C strings at all. The first character of the string is the length – an unsigned value 0 to 255. The next N bytes are the string itself, and there is *no* terminating character. BCPL strings are thus limited to 255 characters. In many cases one talks about BPTRs to BSTRs (BCPL pointers to BCPL strings).

BCPL libraries are the worst – in fact, they are still pretty much an unexplored region. Fortunately, we will not have to deal with them. Suffice it to say that *dos.library* (the C interface to the BCPL AmigaDOS) does some rather nasty stuff to 'glue' AmigaDOS into the operating system.

Although AmigaDOS is admittedly somewhat of a kludge, the concepts behind it are well thought out and quite flexible. The most important aspect of AmigaDOS is that each device driver (such as CON:, DF0: and RAM:) is a separate process. Application programs and DOS communicate by sending *packets* back and forth using (surprise!) Exec message ports. It only *seems* synchronous because the *dos.library* interface, which hides all of these workings, creates and sends the packet on your behalf, and waits for it to be returned from the device driver.

If you send the packet yourself instead of letting DOS do the dirty work, you can do things not possible with the higher-level DOS routines. Two examples are given in this article: asynchronous writing, where data is written to a file while your program continues executing code; and obtaining the pointer to the window used by a process's console.

The general procedure for sending a packet is quite simple:

(A) Find the handler you're going to send the packet to
(B) Construct a DOS packet
(C) Fill in the packet type and argument fields of the packet
(D) Send the packet to the handler process
(E) Wait for the packet to be returned
(F) Examine the return value in dp_Res1

(A) DOS handlers are usually referred to by a pointer to their process's message port. The pr_ConsoleTask and pr_FileSystemTask fields in the process structure point to the message ports of the console handler and the default file system handler for a process. pr_CIS and pr_COS are BPTRs to FileHandle structures. The fh_Type field in the FileHandle points to the message port of the process controlling that file handle. The *dos.library* DeviceProc() function returns a pointer to the message port of the specified handler process. That is:

```
struct MsgPort *port = DeviceProc("DF0:");
```

Note that you cannot get the handler for such devices as CON: or RAW: with DeviceProc(), since multiple CON:/RAW: processes exist.

(B) See the examples below. Construction is trivial but not standard.

(C) See the examples below. The dp_Type field is set to the packet type (ACTION_WRITE, ACTION_INHIBIT, etc.). dp_Arg1,2,3..7 are set to the arguments for that type. For instance, ACTION_READ and ACTION_WRITE take three arguments:

dp_Arg1 = The Fh_Arg1 field in the file handle
dp_Arg2 = A C pointer to the buffer (no alignment restrictions)
dp_Arg3 = the size of the buffer in bytes.

ACTION_INHIBIT is usually sent to DFx: and takes one argument disallowing or allowing the device driver to access the disk.

dp_Arg1 = DOS_TRUE (-1) or DOS_FALSE (0)

There are about two dozen supported packet types. Not all drivers support all packet types.

(D) Sending the packet is done by using PutMsg() to send it to the message port (E) of the handler – quite simple, in fact. If you intend to do the operation synchronously (that is, wait for the packet to be returned), then you can use your own process's message port as the reply port. Otherwise you must use a private reply port.

(F) The result is in dp_Res1 of the packet. In most cases a DOS_FALSE (0) means an error occurred. If an error does occur, the DOS error code is available in dp_Res2 of the packet.

Example #1

This example demonstrates the ability to do asynchronous WRITE operations to an arbitrary file handle. That is, send a WRITE packet to the process handling the FileHandle without waiting for it to be returned. The message interface is not complicated, but requires a slightly convoluted setup in the message structure, which is also demonstrated by this example. There are some DOS limitations that should be taken account of: (1) We must use a private reply port so as not to confuse DOS, and (2) We cannot have more than one packet pending for a given file handle or we will crash the system. The latter is not a bug in DOS, but a limitation in the DOS software. Effectively, this means that if we are doing asynchronous writes to a given file handle, we cannot make any other DOS calls that might use that filehandle.

Remember that the buffer you use in a WriteAsync() call should not be modified until after the next WriteAsync() call. You need two buffers because you are effectively double-buffering your writes: you write to the buffer that is *not* currently being written to the file, then send that buffer and write to the other one. The return value is not available until the next WriteAsync() call, with the EndAsync() call returning the return value for the WRITE begun in the very last WriteAsync() call.

```
/*
 * EXAMPLE1.C  ASYNCHRONOUS WRITING
 *
 * This program demonstrates asynchronous writing using packets by
 * writing to a file while printing to the screen.
 *
 * NOTE: Integers are assumed to be 32 bits long in this example.
 *     The program is run from a CLI or shell.
 */
#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>

#define BTOC(bptr)          ((long)(bptr) << 2)
#define CTOB(cptr)          ((long)(cptr) >> 2)

typedef struct FileHandle       FH;
typedef struct StandardPacket   STDPKT;
typedef struct MsgPort          MSGPORT;
typedef struct Task             TASK;

extern  BPTR  Open();
extern  TASK  *FindTask();
extern  char  *AllocMem();

static  char  Buf1[8192];
static  char  Buf2[4096];

main()
{
    BPTR fh = Open("DF0:test", MODE_NEWFILE); /* Using DF0: on purpose */
    short i;
    long  res;

    if (!fh) {
        puts("Unable to open DF0:test");
        exit(20);
    }
    InitAsync(fh);
                                    /* stuff stuff into Buf1 */
    WriteAsync(Buf1, sizeof(Buf1));
    printf("WriteAsync(buf,%ld) #1 has returned\n", sizeof(Buf1));
                                    /* stuff stuff into Buf2 */
    for (i = 0; i < 10; ++i) {
        puts("Look ma. . . writing and talking at the same time.");
        Delay(10);
    }
    res = WriteAsync(Buf2, sizeof(Buf2));
    printf("WriteAsync(buf,%ld) #2 has returned, result %ld\n",
            sizeof(Buf2), res);
    res = EndAsync();
    printf("EndAsync() has returned, result %ld\n", res);
    Close(fh);
}

/*
 * Asynchronous handler routines. The routines in this example can
 * handle only a single asynchronous file handle. Operation is
 * as follows:
 *
 * (1) Initialize the file handle with InitAsync(fh)
 * (2) Do any number of WriteAsync(buf, bytes) calls
 * (3) UnInitialize the file handle with EndAsync()
 * Do not make any other DOS calls using the filehandle in between.
 *
 * WriteAsync() returns the return value from the previous WRITE
 * packet. NULL can be sent as the buffer causing the routine to
 * wait for the pending asynchronous write (if any) to complete.
 * EndAsync() returns the return value from the last WRITE packet.
 *
 * So, WriteAsync() *does* block, but only if the previous WRITE
 * operation has not completed yet.
 */
static FH          *Fh;        /* The file handle                    */
static STDPKT      *Packet;    /* a standard DOS message & packet    */
                               /* MUST be longword aligned           */
static MSGPORT     *RPort;     /* A standard EXEC message port       */
static char        Pending;    /* operation 'in progress' flag       */

InitAsync(fh)
BPTR fh;
{
    Fh = (FH *)BTOC(fh);
    Packet  = (STDPKT *)AllocMem(sizeof(STDPKT),
            MEMF_CLEAR | MEMF_PUBLIC);
    RPort   = (MSGPORT *)AllocMem(sizeof(MSGPORT),
            MEMF_CLEAR | MEMF_PUBLIC);
    RPort->mp_Node.ln_Type    = NT_MSGPORT;
    RPort->mp_Node.ln_Name    = "Async";
    RPort->mp_Flags           = PA_SIGNAL;
    RPort->mp_SigBit          = AllocSignal(-1);
    RPort->mp_SigTask         = FindTask(NULL);
    NewList(&RPort->mp_MsgList);
}

WriteAsync(buf, bytes)
char *buf;
int bytes;
{
    long result;

    if (Pending) {
        WaitPort(RPort);        /* wait for previous packet to return  */
        GetMsg(RPort);          /* remove packet from port             */
        Pending = 0;
    }
    result = Packet->sp_Pkt.dp_Res1;
    if (buf) {
        /*
```

```
        * DOS requires a pointer to the dos packet be placed
        * in the ln_Name field of the message, and a pointer
        * to the message be placed in the dp_Link field of the
        * dos packet.
        */
        Packet->sp_Msg.mn_Node.ln_Name = (char *)&(Packet->sp_Pkt);
        Packet->sp_Pkt.dp_Link     = &Packet->sp_Msg;
        Packet->sp_Pkt.dp_Port     = RPort;
        Packet->sp_Pkt.dp_Type     = ACTION_WRITE;
        Packet->sp_Pkt.dp_Arg1     = Fh->fh_Arg1;
        Packet->sp_Pkt.dp_Arg2     = (long)buf;
        Packet->sp_Pkt.dp_Arg3     = bytes;
        PutMsg (Fh->fh_Type, Packet);
        Pending = 1;
    }
    return(result);
}


EndAsync()
{
    long result;

    if (Pending) {
        WaitPort(RPort);
        GetMsg(RPort);
        Pending = 0;
    }
    result = Packet->sp_Pkt.dp_Res1;
    FreeSignal(RPort->mp_SigBit);
    FreeMem(RPort, sizeof(MSGPORT));
    FreeMem(Packet, sizeof(STDPKT));
    return(result);
}
```

Example #2

This second example demonstrates how to extract the Intuition Window pointer used by the controlling terminal (a console device) by sending a special packet to the device.

```
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>

#define BTOC(bptr)  ((long)(bptr) << 2)
#define CTOB(cptr)  ((long)(cptr) >> 2)

typedef struct Task              TASK;
typedef struct Process           PROC;
typedef struct StandardPacket    STDPKT;
typedef struct MsgPort           MSGPORT;
typedef struct Window            WIN;
typedef struct InfoData          INFODATA;

extern TASK   *FindTask();
extern char   *AllocMem();
extern WIN    *GetWindow();
extern long   OpenLibrary();

long IntuitionBase, GfxBase;

main()
{
    WIN *win = GetWindow();          /* the meat      */
    int i, j, bt, bb, bl, br;        /* scratch vars  */
    /*
     * Have some fun with the window
     */
    if (!win) {
        puts("No Window!");
        exit(1);
    }
    IntuitionBase = OpenLibrary("intuition.library", 0);
    GfxBase = OpenLibrary("graphics.library", 0);
    if (IntuitionBase && GfxBase) {
        printf("WINDOW = %08lx. Width = %ld Height = %ld\n",
                win, win->Width, win->Height);
        bt = win->BorderTop;
        bb = win->BorderBottom;
        bl = win->BorderLeft;
        br = win->BorderRight;

        for (i = bl; i < win->Width - br; i += 16) {
            j = win->Height-bb - i*(win->Height-bt-bb)/(win->Width-br);
            Move(win->RPort, i, bt);
            Draw(win->RPort, bl, j);
        }
    } else {
        puts("Unable to open libraries");
    }
    if (IntuitionBase)
        CloseLibrary(IntuitionBase);
    if (GfxBase)
        CloseLibrary(GfxBase);
}


/*
 * GETWINDOW()
 *
 * Return the window used by the console of the current process. We can
 * use our process's message port as the reply port since it is a
 * synchronous packet (we wait for the result to come back). WARNING:
 * This routine does not check if the 'console' of the current process
 * is really a console device.
 *
 * The DISK_INFO packet is sent to the console device. Although this
 * packet is normally used to retrieve disk information from disk
 * devices, the console device recognizes the packet and places a pointer
 * to the window in id_VolumeNode of the infodata structure. A pointer
 * to the console unit is also placed in id_InUse of the infodata structure.
 */

WIN *
GetWindow()
{
    PROC      *proc;
    STDPKT    *packet;
    INFODATA  *infodata;
    long      result;
    WIN       *win;

    proc = (PROC *)FindTask(NULL);
    if (!proc->pr_ConsoleTask)
        return(NULL);
    /*
     * NOTE: Since DOS requires the packet and infodata structures to
     * be longword aligned, we cannot declare them globally or on the
     * stack (word aligned). AllocMem() always returns longword
     * aligned pointers.
     */

    packet   = (STDPKT  *)AllocMem(sizeof(STDPKT),
            MEMF_CLEAR|MEMF_PUBLIC);
    infodata = (INFODATA *)AllocMem(sizeof(INFODATA),
            MEMF_CLEAR|MEMF_PUBLIC);

    packet->sp_Msg.mn_Node.ln_Name= (char *)&(packet->sp_Pkt);
    packet->sp_Pkt.dp_Link          = &packet->sp_Msg;
    packet->sp_Pkt.dp_Port          = &proc->pr_MsgPort;
    packet->sp_Pkt.dp_Type          = ACTION_DISK_INFO;
    packet->sp_Pkt.dp_Arg1          = CTOB(infodata);
    PutMsg(proc->pr_ConsoleTask, packet);
    WaitPort(&proc->pr_MsgPort);
    GetMsg(&proc->pr_MsgPort);

    result = packet->sp_Pkt.dp_Res1;
    win = (WIN *)infodata->id_VolumeNode;
    /* note: id_InUse holds a pointer to the console unit also */
    FreeMem(packet, sizeof(STDPKT));
    FreeMem(infodata, sizeof(INFODATA));
    if (!result)
        return(NULL);
    return(win);
}
```

The SCA Amiga Virus

by J.C. Bat

What it is and how to avoid it.

J.C. Bat is currently working as a Software Engineer writing Unix programs in 'C'. He has an Associate's degree in computer science and a strong background in 6502 machine language programming. J.C. has been involved with the Amiga since it was first released, and is currently working on several Amiga products to be released in 1988.

The purpose of this article is to clear up any misunderstandings of the Virus and to enlighten users who may not yet be aware of it. I will cover what this Virus is, what it does, how to tell if you are infected, what to do if you are infected, and how to protect yourself from getting infected.

I first read of the SCA Virus through the posting of Virus-related messages on a local BBS. It sounded like a nightmare come true – something I was surely never to experience (or so I thought). As the messages about the Virus began to increase, so did my interest in just how the Virus works. I had to get a copy of it.

I posted messages asking for someone to send me a copy of the Virus. Time passed and no disk arrived. This lead me wrongly to believe that the Virus was not a very common thing. In the meantime, a friend of mine, Rick, started having problems with his Amiga – his new games would play one time and not load or play again. The local dealer was getting tired of him returning his disks. I looked at his Amiga and could not find anything wrong. Then one Sunday night Rick called:

Rick: Are you still looking for the Virus?
Me: Yeah, sure. Did you get a copy of it?
Rick: Copy of it! I'm infected with the #%↑!@ thing!
Me: What do mean infected?
Rick: Just about every one of my disks has it!

Now I was armed with a copy of the Virus, and ready to explore just how it works. Several weeks of work later, here are the results of my Virus research.

Where did this Virus come from?

The Virus was originally written in Switzerland by a group of hackers who break protection routines on commercial programs. SCA is short for Swiss Cracking Association. No one knows when, where or how the Virus came to North America, but it is believed to have arrived on one of these SCA broken copies. I have heard from very reliable sources that the Virus has now spread to the point where it has been shipped on disks from several commercial software producers here and in Europe. So what started as a group of hackers showing off their programming skills has turned into a real nightmare for all Amiga users worldwide. I hope that a better understanding of how the Virus works will help stop it from spreading further.

Where does this Virus live?

This Virus exists in a modified version of the Amiga's boot blocks on a bootable DOS disk (Workbench). The Virus actually replaces the normal boot code with its own infected code. These boot blocks are the first two sectors on a DOS disk (Cylinder 0, Sectors 0 & 1, Head 0). Before booting, the Amiga reads these two sectors to see if the disk is a DOS disk. The sectors contain the required DOS initialization code, and there is enough unused room in them for other special information (like copy protection routines). The Amiga then copies this DOS information into memory. The original DOS code opens the CLI window that all Amiga disks use during boot (the one that shows you the copyright information and version numbers). The Virus has additional code beyond the normal DOS startup routines; this added code is what does the infecting of other disks, and also displays the Virus message. The infected boot blocks are located in memory beginning at $7EC00.

What does this Virus do?

The first thing the Virus does is protect itself in memory so that it will not be deleted during warm starts (Control-Amiga-Amiga). Yes, I do mean that this program can survive a reset. After a warm start is when the Virus code does its dirty little deeds. Its goal is to infect (copy itself onto) each disk that your Amiga is warm-started from. If the disk is write protected, the Virus cannot copy itself and no harm is done. In addition to copying itself onto other disks, the Virus also has a counter that keeps track of how many disks it has infected. When it reaches a certain number, it will break out of the normal boot sequence and enter into a special graphics routine. This graphics routine will display a black screen with horizontally scrolling text across the centre of the screen. When the Virus is finished displaying this text, it proceeds with the boot sequence as normal and the counter is reset to zero.

How do I detect this Virus?

The Virus can be detected both on disk and in memory. To detect the Virus on disk you can use a public domain utility called "Disk Zap" (Fish #50) that will allow you to examine the boot sectors. An infected disk is easy to spot, just search for this text string:

Something wonderful has happened Your AMIGA is alive!!!
and even better Some of your disks are infected by a Virus
Another masterpiece of the Mega-Mighty SCA

(This is the text that is displayed when the infection counter is triggered.)

Another way to check for the Virus on disk is with the utility "VCheck1.0". VCheck is very convenient – you simply enter "VCheck" at the CLI prompt followed by the drive number to check. VCheck then reads the disk looking for the above text. You are informed of your disk's condition, infected or healthy. VCheck is in the public domain.

Checking memory for the Virus can be done by looking at the memory starting $7EC00 and searching for the above text. I'm not aware of any public domain utilities to dump memory, but several commercial programs do allow this. There is a public domain program that checks the $7EC00 area of memory for the above text called VirusTest. One other method of checking for (and also removing) the Virus in memory, is to warm boot your Amiga while holding the left mouse button down. If the screen turns green, and stays green for several seconds, your Amiga had the Virus in memory. It has now been removed. This is a back door that the original SCA Virus programmers added for their own use, and now you can use it too.

VCheck, VirusTest, and Disk Zap can be found on most of the commercial database services or on a BBS service. If you can not find these programs, they are available in one ARC file called Virus.ARC from the Berks Amiga BBS at (215) 678-7691 – 300/1200 baud – 24 hours. (This ARC file will also be available on the first *Transactor for the Amiga* disk, released concurrently with this issue. –NS).

How do I get rid of this Virus?

Getting rid of the Virus is the easy part. Simply "INSTALL" an infected disk to remove it. Install will write clean AmigaDOS code over the Virus code on your disk. Easy enough? Up until now the Virus has looked like a harmless programmer's trick. Well, a word of warning: remember back at the beginning of this article when I said that Rick's new game disks would load the first time and then not load again? Remember when I told you that the extra space on the boot sectors could be used for copy protection routines? Well, when the Virus spreads itself to those disks, the copy protection routines are overwritten and future checks for the protection fail. The result is a disk that will not load.

So, if you have a disk that is infected and it loads, doing an Install on that disk will remove the Virus. Your disk will now be clean and should load just like before. If you have a disk that has the Virus and will not load, installing the disk will remove the Virus but odds are the disk will still not load because some part of the protection has been destroyed by the Virus. Are we having fun yet?

How do I protect against this Virus?

1) Never warm boot your Amiga, always cold start (power off – then back on) your Amiga. Not real practical, but very effective.
2) Hold down the left mouse button each time you warm boot. Seems easy enough; only problem is if you forget one time. . .
3) Use VCheck on every bootable disk before you boot it.
4) Always write protect your disks before booting.
5) Use VirusTest regularly.

I feel the best protection is a combination of these tips. I try never to boot from any disk except my personal boot disk. I try to hold the left mouse button down during every warm start. I always check *every* disk I receive before I boot from it, and I never boot from a disk that is not write protected.

I want to see the Virus message!

IF you are one of those who just has to see the SCA Virus message, follow these guidelines for building a controlled Virus test disk.

1) Format a new disk
2) Create an "S" directory
3) Install this new disk
4) Copy "Install", "Echo", and "VCheck" to the root directory
5) Enter this startup-sequence in the "S" directory

```
vcheck df0:
install df0:
vcheck df0:
echo 'ReBoot Me!'
```

6) Boot your Amiga from an infected disk
7) Reboot from your new test disk

Now keep rebooting your Amiga when requested. What you are actually doing is booting with a clean disk, infecting it, then deleting the Virus from disk, and finally starting the process over again. So keep on rebooting until you see the Virus message. Remember to format this test disk when done!

I have written this article with the goal of education so that you, the educated Amiga user, can help to stomp out this Virus. Please do not violate my trust and use any of this information to develop or promote future strains of Virus. If we are all vigilant, this blemish can be removed from the face of the Amiga – "Spread the word, not the Virus"! Tell everyone you can, share your knowledge with others. Remember, only Amiga makes it possible, and only you make the Virus possible.

SCA VIRUS Update

Since the article above was written, a new strain of the SCA Virus has surfaced. It is nothing more than the original SCA Virus with the text message portion altered. This has the effect of defeating VCheck1.0, VCheck1.1 and VirusCheck. A new version of VCheck has been released to deal with this new strain. It is called VCheck1.2 and should be available in the public domain by the time you read this. I have also added VCheck1.2 to the VIRUS.ARC file referred to in my original article. Changes in VCheck1.2 include the ability to check for both versions of the Virus, and also to check your Amiga's memory for the Virus. My personal fear is that other versions of the Virus text will eventually appear (No, I'm not going to tell you how!), resulting in the release of many more VCheck-immune "new strains" of the SCA Virus. When you consider the amount of time it takes for a new version of the Virus to surface, the time it takes for VCheck to be updated, and the time it takes for VCheck to be released and circulated, you can see that Amiga users could be facing a major problem. For that reason, I now suggest that DiskZap be used as your number one defence against all current and future strains of the SCA Virus. DiskZap is easy to use and, most importantly, since you are the actual testing device, you will not be easily defeated by sick-minded individuals altering the text string(s).

J.C. Bat, December 12, 1987

Break\Point

by Victor A. Wagner, Jr.

First in a series on the art and science of software debugging. In this instalment, a look at the bad old days. . .

Vic Wagner started his association with computers in 1965 while in the US Air Force working in a group studying digital flight simulation. Since becoming a civilian in '66, he has been working mostly with minicomputer manufacturers in the real-time systems software area. Weekdays from 8 to 5 find Vic answering the technical support line for Computer Automation Inc., and maintaining three real-time software systems. Evenings and weekends he spends most of his waking hours talking to Taarna (his Amiga 1000), writing programs, and spending time on CIS AmigaForum (76046,3004). Vic is probably best known in the Amiga area for his in-depth knowledge of the well-known debugging program MetaScope, published by Metadigm, Inc.

'Twas the night before Beta
And all through the house
Not a creature was stirring
Not even my mouse.
The breakpoints were set
Through the program with care
For fear that the Guru
Soon would be there.

If my mouse isn't moving, it's almost a sure sign that the Guru will visit any moment – never mind how careful I've been. Well, it looks like I'm going to have to start this debugging session all over, so let's start at the beginning.

Even before we get into the nitty-gritty details of debugging on the Amiga, though, I would like to review a little computer history. Understanding where we came from may help us understand where we are, and why we are here. Some of the techniques used in the early days are still applicable. Others, while still do-able, seem to be mostly a waste of time (and perhaps they are).

First of all, we have the word itself: "Debug". Where did this word come from? I heard this story around 20 years ago, and recently saw an interview with Admiral Grace Hopper, who told the same story, so perhaps it *is* true.

It seems that one of the early computers had been working quite well for some time, when suddenly it began to behave erratically. Sometimes test programs would work properly, other times not. Those working with the machine were figuratively tearing their hair out. The source of the erratic behaviour was ultimately discovered: a moth had got caught in one of the relays (this was an *early* computer), causing it to jam. Since removing a bug from the computer fixed the problem, "debugging" a computer was born.

Well, if all it took to fix a computer were a large can of Raid and the patience to crawl around inside the machine with a vacuum cleaner to clear out the dead bugs, debugging would today be a minor specialty of the insect control profession, not mentioned in polite company. Debuggers would probably become pariahs because of the permanent stench of pesticide. There also wouldn't be any more of this article. But a lot of things have changed since the first computer was debugged. For one, computers are now too small to crawl around in (even the moths and smaller insects can't get in). And a much larger change occurred with the advent of 'software'.

The original computers were 'hard-wired'. They were built to do but one job. If they weren't doing it, then some 'malfunction' had occurred and the machine needed to be repaired. Finding the broken component could be (and sometimes was) a long and tedious job. Tools were invented for the purpose of finding these malfunctions. The ubiquitous oscilloscope and the more recent logic analyzer are familiar tools of the trade.

The Birth of Software

Soon it became clear that building a new computer each time a new problem needed solving was going to be an enormous drain on the resources of the human race. It was pointed out that instead of storing only data in the memory of the computer, it would be possible to store representations of parts of the algorithm in the memory. What we call a computer today was called, in the early days, a stored program digital computer. With the advent of the stored program digital computer, another class of errors came into existence: software errors.

There is a *big* difference between the job of finding hardware errors and that of finding software errors. When a hardware debugger goes to search for a malfunction, he *knows* that the hardware was functioning at some time in the past. That is, something that used to work is now broken. Software, on the other hand, doesn't break. If there is a bug in the software, it has been there all along. Another distinction is the relative complex-

ity of the objects being debugged. My experience as both a hardware and software 'fixer-upper' leads me to the conclusion that software now far exceeds hardware in complexity. We need some tools.

When the first computer programs were written, there were no compilers. There weren't even any assemblers. Programmers figured out what bits they wanted in the memory and painstakingly (sometimes by hand with toggle switches) put them there. They then ran the programs and compared the output with some laboriously hand-generated answers. If they were lucky, everything worked as planned. If there was an error, they would look through memory (ensuring that all of the instructions were correct). If that failed, then perhaps they would force the machine to execute one instruction at a time (recording all the intermediate results) and attempt to find the problem that way. But single-stepping a computer meant lots of down-time, and that was a costly proposition – early computers were very, very expensive. Ways had to be found to allow programmers to debug without tying up the machine.

The memory dump (a listing of the entire contents of memory on a printer) was the first debugging tool. These dumps could be produced relatively cheaply (don't forget that a 16 kilobyte machine was a big computer in those days). Armed with such a dump, the programmer would then spend hours (sometimes days) at his desk trying to find the problem. The technique is still used in some places: poring over core dumps has accounted for some of the most tedious debugging time I have spent.

It turns out that debugging really wasn't much of a problem back in the beginning – it took so much effort to produce a program that there weren't very many which needed debugging. As a matter of fact, *so* much effort was spent in creating a program that debugging wasn't even recognized as a separate phase. It was viewed more as a verification of the correctness of the design and implementation.

The big problem then was to get programs created more efficiently.

Along Comes the Assembler

Some bright person, I don't know the name, observed that computers were very good at doing methodical, tedious things. They also had very good memories. Out of these rather unremarkable observations came a rather remarkable thing: the assembler. The assembler allowed the programmer to concentrate on what the computer should do, rather than on trying to remember that, for example, an ADD instruction had bits 0,3,4,7 turned on. When someone else observed that names could be given to memory locations (and other things as well), the symbolic assembler sprang forth. Suddenly, a plethora of programs was being created. Debugging still wasn't considered a really serious problem, though, because these early programs were rather simple in comparison to those of today. Simple programs can often be debugged simply by perusing the listing. Failing that, perhaps having someone else examine the listing will suffice. And if *that* fails, well, there is always the memory dump, late hours and gallons of coffee as a last resort.

Some programs available at the time foreshadowed what we now think of as a debugger. These programs allowed the inspection and modification of memory locations (usually in several formats). While these programs certainly had their place, it is doubtful that anyone these days would recognize them as debuggers. It is important to keep in mind that magnetic storage was not then readily available. Most program source was kept either on paper tape or cards. Executable programs were more likely to be kept on paper tape, since card punches were very expensive.

Keeping binary programs on paper tape wasn't too bad, but consider the problems associated with source on paper tape. Editing the source of a paper tape program was expensive: the tape needed to be copied with the changes inserted (at least with cards you could just replace some cards or insert a few). This creates a new tape that is around the same size as the last one. Programmers were loath to make source changes every time a bug was found because of the massive amounts of paper tape that would soon be cluttering their offices. Instead, they 'patched' the executable with their handy utility routine and made a new executable with the same utility. One of the more important features of these utilities was the ability to write memory onto paper tape (or cards) in a form suitable for reloading once it had been modified with the new patch.

The typical 'latest' copy of a program was generally a source paper tape (or card deck) of some recent vintage, a marked-up source listing (produced by the assembler) with 'LATEST' scrawled boldly across page one with a magic marker, and maybe the last two or three versions of the patched executable.

When the listing got too ragged or we could no longer readily make out the scrawls in the margins, we reluctantly edited our source and re-assembled.

Of course we marked the new listing with 'LATEST' in large bold friendly letters. You are probably wondering why we didn't use the date at the top of the listing to keep track of the latest. Well, back then there weren't many operating systems – assemblers were generally stand-alone programs. You cold-started the machine, then loaded and executed the assembler. This meant that there were no dates at the top of the listings. If you're wondering why we didn't write the date on by hand, all I can answer is "I don't know".

A typical debugging session went something like this:

1) Get just-assembled listing and object tape.

2) Get objects for each of the 'library' subroutines we're going to use.

   This may require some explanation. Linkers were rare back then, so each program or subroutine was assembled at an absolute memory location. The assignment of these addresses was left up to the project manager, who kept a 'memory map' showing where each of the pieces would fit, as well as the entry points for all the routines and the argument requirements and resource usage of each.

3) Load the utility program, then with the utility program load all of the pieces to make up the whole program.

4) If you are smart, make a new tape of the current memory configuration before something blows it up.

5) Set up any additional memory that may be required with the utility program.

6) If the machine you're working on has a hardware 'Address Halt', set it to the address where you wish to stop the machine and look around.

   If the machine doesn't have 'Address Halt', put a halt instruction where you wish to stop.

7) Jump to the address where you wish to start execution of the program and wait for it to halt.

8) If your machine doesn't have 'Address Halt' replace the instruction that got clobbered when you inserted the halt instruction.

9) Look at all the registers (all machines had consoles on which you could examine the contents of the registers), perhaps examine a few memory locations, and decide whether to go on, or start over.

10) Many machines had a 'Single Step' so that the computer would execute only one instruction at a time. "Going on" in many cases consisted of 'single stepping' and watching the registers to verify the program's operation.

11) Upon finding an error in the program, go to the start address of the utility program, patch memory, and return to step 4. If your time is almost up on the machine, collect your stuff and leave. Oh yes, don't forget to write down the patches you just made to memory on the listing, so that some day you can edit the source and re-assemble.

Single Step – A Giant Leap?

Around this time several things happened. I don't know which came first, so we'll just have to guess. Someone noticed that a lot of testing/debugging time was spent pushing the 'single step' button, then recording the registers, then pushing the 'single step'. . . ad nauseam. Since this is obviously a repetitive task, and computers are good at repetitive tasks, a program was written that mimicked this behavior. This program actually simulated the execution of each instruction and printed the register contents after each simulated instruction was executed.

Many forms of this 'Trace' program appeared, with all sorts of different features. Some would display the instruction being traced in the same format used by the assembler recognized. Some would display only those registers that had changed. Many could be told which addresses or address range was important and would only print trace information in them. Trace programs became more popular than hula hoops. Managers loved them. No longer would programmers be sitting at the console taking up valuable computer time (the average computer back then cost at least as much as ten programmers for a year, and upkeep and operations were costing around four programmers' worth). The typical debugging session now became: load program, set up trace stuff, watch line-printer print out several hundred pages, pick up stuff and go back to desk.

Since these trace programs couldn't make decisions based on what was found at some point in their execution, it is my opinion that a *lot* of computer time and a *lot* of printer paper were wasted. My personal experience with them was that either they did me no good at all (what I needed to see wasn't in the trace) or it was there but on page one or two out of several hundred. Walking out of the computer room with several hundred pages of trace output did look impressive, though, and the hours you spent marking them up copiously in red and black did make you look very busy. And you were – but I'm not sure it was productive time.

Maybe I just never learned how to properly use the trace programs. Maybe I just never had a really good one; I don't know. I do know that I saw actual instances of trace programs tracing the trace program tracing. . . well, you get the picture. Boy, is that a waste of paper – and computer time! At any rate, it kept the programmers out of trouble and undoubtedly pumped lots of money into the economy. I'm sure that both the printer manufacturers and printer supply manufacturers *loved* trace programs.

Another Innovation: Relocatability

Also around this time, a new word began to make its way into the consciousness of the computer world. A new buzzword: *relocatable*. Notice how it rolls off the tongue. . . RE-lo-CA-table. Nice ring to it.

Unless you were in the know, it meant nothing to you. Non-programmers marvelled at all this 'magic' stuff that the programmers wandered around talking about. This was wonderful for most programmers' egos, and many of us got rather swelled heads. The stereotype of the prima-donna programmer wasn't far off in many cases.

Anyway, while Relocatability allowed a couple of things to happen to make *writing* programs much easier, its effect on testing was not such a smashing success. I mentioned earlier that assemblies were done with absolute addresses; that is, the address of each component of the assembly was known at the time the assembler was executed. One of the problems of mapping out the memory was that, every once in a while, one of the pieces would get too large for the space allocated to it, so some or all of the pieces would have to be moved. When this happened, those pieces that had been referencing the parts that had been moved would have to be re-assembled. Lots of busy work.

Relocatable code was supposed to solve all of those problems, and to some extent it did. I must be fair and point out that the address allocation problem did go away, but there was a price.

In order to use this whiz-bang new feature, all we needed was:

1) a new relocatable assembler; and,
2) a relocatable *linking* loader.

What is this *linking* stuff? Well, when the address of all of the parts of a system were known, we could just put a statement in the assembler that said "the trigonometric sine routine is at address 123", which told the assembler how to call the sine routine. This generally looked something like:

```
SINE    EQU         123
```

It was, of course, the programmer's responsibility to see that the sine routine actually *was* at address 123 after the program was loaded into memory. Well, now we had a different method for telling the assembler where to find the sine routine.

It looked something like:

```
EXTERNAL    SINE
```

This told the assembler to enter into a conspiracy with the "relocatable linking loader" to ensure that the correct address of SINE would be inserted in the code in all the appropriate places so that instructions referring to SINE would execute properly. This is all well and good, but how does the loader know where the SINE routine is? Well, in the assembly that contained SINE we would find a statement that looked something like this:

```
NAME        SINE
```

and, later on somewhere in the same program:

```
SINE    EQU         $
```

As an aside, for those of you who don't know why "$" was used to represent the current location, it is a pun. Somebody somewhere was writing an assembler and needed a symbol for the current location. "Current" → "currency" → currency symbol → "$" (at least in the US).

Anyway, back to relocatable linking loaders: Now the loader can find where SINE is and can 'fix' all of the programs that refer to it. A really smart loader can even decide whether a particular assembly module is required, based upon whether any of its NAMEd entities are referenced elsewhere. So, we can load the program, and we have neatly avoided the problems associated with absolute addressing. Now, how do we find out where everything is? Our assembly listing, which used to have nice addresses in it, now starts at 0 (which is almost certainly *not* where the program is loaded).

The answer to this problem was a document known as a 'load map', generated by the loader, listing all the symbols and the addresses that had been assigned to them. If we were lucky, the list would be available both in numerical and alphabetical order. At this point we went back to the same old process of using the utility program to load, 'single step' and look, or using the trace program (after doing all the arithmetic to determine where location 123 in my program really is in memory).

The construction of programs had certainly become easier, but testing was about the same as it had always been – except that we all got very good at octal arithmetic (yes *octal* – everyone used it in those days). We were getting there, but we hadn't quite arrived.

Now that programs were easier to write, more computers were built, more programs were written, and so on. There was an explosion of information processing equipment and programs. Debugging still wasn't perceived as a major problem, though, not compared to the whole problem of getting a program done: Design, Implementation, and test. (Yes, "test" was always spelled with a small letter; still is, in most places.) People believed that if the expression of the problem could somehow be made easier, then testing would go away. It was felt that the big problem lay not in program design, but in the mistakes made by error-prone human beings transcribing the design into those funny little instructions the computer needed in order to do its job.

And Finally, a High Level Language

Another very bright person accordingly came up with the idea of writing a program that would translate mathematical expressions into computer code automatically.

Since almost all the programs then being written were heavy on the number crunching, this seemed like a very good idea. Out of it came FORTRAN, which stands for, of course, FORmula TRANslator. Wow, we could write lots of programs now! We could write programs very quickly now! We wouldn't have to worry about the design not being implemented correctly – the computer would generate the correct instructions all by itself!

WOW!

As you've probably surmised, this didn't really solve the problem. It merely changed the perception of where the problem really was. Some people actually started spelling "Test" with a capital letter. Some others began writing programs that would help the others to find the problems in their programs. Hardware was added to some computers to facilitate Testing. The light was beginning to dawn. Anyone could write a program, but writing a *correct* program was something else again.

We'll get into some details of actually debugging next time. Until then, here's a question: What technique for debugging became practical with the introduction of FORTRAN? This technique is still in use today and works on all machines with HLLs. Another hint: We've *all* used this technique.

Next month, we'll learn about the breakpoint, after which this article is named, and see what a profound effect it had on debugging technique. We will also learn what a radical change the CRT made to programming (Design, Implementation. . . and Test).

$13 Common Amiga Programmer's Errors . . . and how to avoid them

by Bryce Nesbitt

Bryce Nesbitt is a freelance hacker from Berkeley, California who has worked on many commercial projects for Commodore microcomputers. Some of his best known achievements are the 1541 Flash and the Quicksilver IEEE Interface, both sold by Skyles Electric Works. His Amiga contributions include the program PopToFront, which appeared in Transactor, Volume 8, Issue 1.

This is a countdown of $13 of the most common errors that I have noticed in commercial software for the Amiga. The offenders range from basement software houses up to multinational corporations. Keeping these problem areas in mind can greatly improve your software.

$13> Some programs randomly trash the lowest 50 or so bytes of memory. This can cause all sort of subtle headaches that might not be easy to track down. The situation is commonly caused by referencing from a pointer variable that was never properly initialized, and instead ended up with the value of zero.

A publicly available program called **MemWatch** (by John Toebes of The Software Distillery) can be set to sit in the background continually scanning low memory. If it detects illegal access it will post a warning. (Memwatch is available from Fish disk #48 or the Transactor Amiga Disk for this issue.)

Simply setting location zero to an offbeat value such as $00DFF001 will cause some software to fail.

$12> By default the DOS will place requesters relating to your process in the Workbench screen. If your program opens a custom screen it must direct DOS to the new location. This code fragment shows how to do that:

```
#include "libraries/dosextens.h"
    . . .
struct Process      *mproc;
struct Window       *window;
APTR                temp;

    . . .
    mproc  = (struct Process *)FindTask(0L);
    temp   = mproc->pr_WindowPtr;              /* save old value */
    mproc->pr_WindowPtr = (APTR)window;        /* set a pointer to any open
                                                 * window on your screen */
    . . .
    mproc->pr_WindowPtr = temp; /* restore value before CloseWindow */
    . . .
    CloseWindow(window);
```

See the *AmigaDOS Technical Reference Manual* for more details.

$11> Dire warnings should accompany all of the Intuition "VERIFY" messages. These turn out to be tricky to use under certain circumstances. Read the *Amiga Enhancer Software Manual* for details. One of the conditions is important enough to mention here: while a VERIFY function is enabled, *never* call AutoRequest() or make any call that might indirectly or otherwise make a DOS call – this includes OpenLibrary(), OpenDevice() and OpenDiskFont().

$10> If you process **RAWKEY** messages your program is only half baked! Raw keycodes provide positional information, not actual key values. Each country the Amiga is sold in has its own keymap. The raw codes must be "cooked" to find out what they really mean. This involves two steps:

1) Open the console.device and extract its "library" pointer. Other than the odd method of extraction, the pointer works just like that of a normal library.

Warning: Some versions of the Aztec C "functions.h" file have a bug that causes a duplicate symbol error. To correct it, simply remove the line that has the word "ConsoleDevice()".

2) Send all of your RAWKEY IntuiMessages to the DeadKeyConvert() function.

```
struct ConsoleDevice *ConsoleDevice;  /* "library" pointer            */
struct IOStdReq      ioreq;           /* empty, zeroed IOStdReq       */
. . .
    /* Open a console with no Windows or ports attached */
    if ( OpenDevice("console.device", -1L, &ioreq, 0L) )
        exit(21);
    ConsoleDevice = (struct ConsoleDevice *)ioreq.io_Device;
. . .
/*
 * DeadKeyConvert(). From the Amiga Enhancer Manual.
 *
 * Takes an Intuition RAWKEY message and processes it.
 * The result will be "cooked" keys and key conversions in
 * your buffer.
 *
 * Returns -2 if not RAWKEY, -1 if the buffer overflowed, or
 * the number of characters converted if all was ok.
 */

long    DeadKeyConvert(msg, kbuffer, kbsize, kmap)
struct  IntuiMessage    *msg;     /* IntuiMessage to be converted     */
UBYTE                   *kbuffer; /* buffer to be filled              */
long                    kbsize;   /* how large the buffer is          */
struct  KeyMap          *kmap;    /* a custom keymap pointer or zero  */
{
static struct InputEvent ievent = {NULL,IECLASS_RAWKEY,0,0,0};

    if (msg->Class != RAWKEY) return(-2);

    /* pack IntuiMessage into an input event */
    ievent.ie_Code      = msg->Code;
    ievent.ie_Qualifier = msg->Qualifier;
    /* Get previous codes from location pointed to by IAddress
     * This "magic" pointer is valid until the IntuiMessage is
     * replied
     */
    ievent.ie_position.ie_addr = *((APTR*)msg->IAddress);

    return( RawKeyConvert(&ievent,buffer,kbsize,kmap) );
}
```

$F> Know Thy Memory! There are two types of Amiga memory, and three attributes. It is critical that these be used properly.

CHIP type is any memory the custom chips *can* access. Some examples of things that must be allocated from CHIP are:

sprite data
image data (this includes the images for gadgets)
audio buffers
trackdisk device buffers

If you do not remember to *specify* chip for any of the above, your program will fail on an expanded machine. A program that has data for gadgets or pictures that is part of a data segment *must* ensure that this ends up in the correct class of memory. This can be done in five ways:

1) Copy the data into some newly allocated CHIP memory.
2) Use a language that supports specifying memory types for data.
3) For Lattice C users, take advantage of the **Atom** utility.
4) For Manx C users, link with the +Cd option, which forces all data segments to be loaded into CHIP memory.
5) For doing a hack job on existing programs, use the **FixHunk** utility from Fish disk #36 or *Transactor for the Amiga* Disk #1 (containing the programs from this issue).

FAST is the type for any memory the custom chips *can't* access. When present, FAST becomes the default location for your code, data, stack, AllocMem() calls and just about anything else that does not need CHIP. Since some Amigas don't have FAST memory, and since it is the default anyway, you should never actually specify FAST.

PUBLIC is perhaps the most misunderstood attribute. It may be combined with any other flags, and specifies that the memory must not be swapped, mapped out or otherwise be made non-addressable. Since the first generation Amigas don't have the capability to do any of the above to memory, this bit is for future compatibility. Any memory that is used by interrupts or that will be accessed by multiple tasks needs to be allocated with this bit set. Remember that devices are separate tasks, so I/O requests and any memory used for communication with devices qualifies.

Note: Since your program stack is private memory, it is illegal to use the stack to temporarily allocate a structure that requires PUBLIC memory.

CLEAR specifies that the range of memory be cleared to zero before return. It may be combined with any of the other flags.

LARGEST is an attribute useful only for AvailMem() calls. It will return the size of the largest currently free block of memory. (Due to multitasking, the returned value will be only a hint as to how large a block could *actually* be allocated).

Examples:

```
AllocMem(MEMF_CHIP | MEMF_CLEAR | MEMF_PUBLIC);
/* Get public chip memory and clear it before returning */

AllocMem(MEMF_CLEAR); /* Get any type and clear it before return */

AllocMem(0L);         /* Not picky at all, return any type */
```

$E> Keep up with the bugs:

- **WindowToFront()** while the user is holding an icon locks up the machine.
- **ScrollRaster()** left or right in a SuperBitmap window eats memory if there is no previous TmpRas.
- The **serial.device** will crash if characters arrive at the serial port at a vastly different baud rate than expected.
- Fiddling with **rp_Mask** must be done while layers are locked.
- Calling **Delay(0)** or **WaitForChar(X,0)** will occasionally crash.
- **argc** and **argv** are *reversed* for programs launched from the Workbench when linking with some versions of **Astartup.obj**.
- The semaphore functions have defective C language bindings. An example of a workaround is available from Commodore.
- The output of the **Text()** command is clipped to the width of the rastport even if the write was started from beyond the left edge of the rastport.
- If you are using **ReportMouse()** watch out! The order of the arguments depends on what language you are using:

  Aztec C – ReportMouse(Window,Boolean);
  Lattice C – ReportMouse(Boolean,Window);
  Assembler – Window in d0, Boolean in a0.

- CINIT and UCopperListInit() will *not* allocate a UCopList if passed a zero pointer. You must take care of allocating a 12-byte MEMF_CHIP | MEMF_PUBLIC | MEMF_CLEAR buffer and pass the pointer. However, FreeVPortCopLists() will deallocate that memory for you.

$D> To be compatible with Amigas that have a 68010, 68020 or 68030 processor be sure that you:

- Never use the 68000 instruction "MOVE <SR>,EA". If you need to examine the processor condition codes, the Exec function **GetCC()** does this in a compatible manner.
- Don't trash the upper 8 bits of a 32 bit pointer.
- Remember that each of the 68000 family members supports a different stack frame: don't make any assumptions about stack format without checking what CPU is in use.
- Don't write self-modifying code. This will break future systems in multiple ways.

$C> You can increase the apparent reliability of your programs by never running the system down to zero free bytes of memory. An easy way to add some breathing room is to globally replace all your AllocMem() calls with calls to the SafeAllocMem() function given below.

If you need to tell the user that the system is out of memory, but there is not enough memory to post a requester or Alert, simply set the title of your Window or Screen to "** NO MEMORY **" (possibly with a red pen colour). This operation does not require any dynamic allocations.

```
#include "exec/memory.h"
...
#define PANIC_FACTOR_CHIP 25000L
...
APTR SafeAllocMem(size,flags)
long  size;
long  flags;
{
register APTR p;

   if ( p = (APTR)AllocMem(size, flags) )
      if ( AvailMem(MEMF_CHIP) < PANIC_FACTOR_CHIP ) {
         FreeMem(p,size);
         return(0);
      } /* System is low. . . no go! */
   return(p);
}
```

$B> Watch out for return values! Very carefully! Programs will probably crash out if an error return is ignored, and/or the response to the error condition does not work as planned. Bugs of this type are all too common, especially with memory allocation.

$A> Be kind to those who use your program: Put at least one pixel worth of drag bars and page gadgets on all your custom screens.

Implement the **HELP** key, especially if you already have on-line help. If you open a string gadget for the user to type into, use the V1.2 **ActivateGadget()** function to save the user the confusion of clicking into it first. If you normally program with the stack set to something other than the default size, make sure to check that your program will still operate with the default stack. Emulate the position of **AutoRequest()** gadgets: "Cancel" or "No" on the right, "Retry" or "Yes" on the left. Try running your program with the font set from Preferences to **Topaz-60**. If the program can't deal with whatever the default system font might be, at least *specify* a font or set of font attributes that will work.

$9> If you are only interested in the *current* position of the mouse, you should not respond to each mouse event in turn. Instead, skip ahead to the most recent. A good example of this can be found in the chapter of the Intuition manual that describes the IDCMP.

$8> If you get a message of a type, class or code that you don't understand, ignore it. Don't make the assumption that if something is not A or B that it must be C; someone may add new meaning at a later date and blow your application out of the water.

$7> The Amiga Operating System is constantly evolving. If you use any of the newer features you have a responsibility to check the version number. Here is a complete list:

0 ; any
30 ; V1.0
31 ; V1.1
32 ; V1.1 (with PAL support)
33 ; V1.2
34 ; V1.21 (with hard disk/network boot support)

The simplest way to ensure that you are running on a recent enough version of the OS is to specify the required minimum when opening libraries. Since V1.1 Kickstart is completely obsolete, it is best to specify at least version 33 in any new program.

$6> If you open a custom screen, consider that screen width and height of the screen will depend on the user's configuration. The **morerows** utility can be used on any Amiga to increase the available screen area. **PAL** machines have a display that is inherently 56 lines taller than NTSC. And finally, future Amigas will support even higher resolutions.

Under V1.2 there are several options for taking advantage of any extra space that might be available:

1) When opening the screen, set NewScreen.Height to the constant **STDSCREENHEIGHT**. This automagically sets the screen height to the proper maximum. You will need to read the Screen structure to see how high your screen actually ended up.
2) The public variables **GfxBase->NormalDisplayRows** and **NormalDisplayColumns** contain the size of a non-interlace Workbench screen. This has the advantage of allowing the width of your custom screen to be extended to match the Workbench, but be aware that if you extend the width beyond 640 you are not guaranteed to have any available sprites (other than the pointer).
   If it's important, you can also discover the aspect ratio of the display by looking at **GfxBase->NormalDPMX** and **NormalDPMY**.
3) The Intuition function **GetScreenData()** will copy the scoop on the Workbench screen into a buffer. You could then borrow the size variables for use in your own screen open.

GetScreenData() can also be used to open windows that don't obscure the title bar. Easiest is to subtract the height of your window from the height of the screen, and open your window at that location. At the very least, open your windows one pixel *down* from the top of the screen to allow limited access to the screen's drag bar and page gadgets.

Morerows is available from Fish disk #54 or *Transactor for the Amiga* disk #1.

$5> It is impossible to create an **AutoRequest()** that has only a positive response. Specifying zero for NegativeText is a *bug*.

$4> The **MaxWidth** and **MaxHeight** fields of **NewWindow** structures are often set to arbitrary constants such as 640 and 200. This is often *wrong*. Expanded displays are possible, even common, among Amiga owners. Set these fields to the maximum your application can support; even if that is greater than the largest screen you have ever seen. If there is no reason to limit window growth, set both to -1 (or ~0).

$3> Use the copy of **pm** that comes on the V1.2 **Extras** disk. A process monitor will instantly show if your program is hogging CPU when it should be waiting for some external event.

$2> There is a refresh control flag in **NewWindow** structures that should be set in many cases. Even **SMART_REFRESH** windows may generate refresh events if there is a sizing gadget. If you don't have specific code to handle this, you *must* set the **NOCAREREFRESH** flag. If you do have refresh code, be sure to use the **Begin/EndRefresh()** calls. Failure to do one or the other will leave Intuition in an intermediate state, and slow down operation for *all* windows on the screen.

The popular "PowerWindows" utility is a prime culprit in creating windows that should have had this bit set.

$1> Far too many programs have memory leaks. Try this test: run your program, then exit. Check how much memory is free. Now run your program again, then exit. Now check that the amount of free memory is *exactly* the same. If there is *any* difference you have a bug. (If you invoke the Workbench tool with "LoadWB -debug" you get an extra invisible menu item that can be of great use in subtracting out the effects of resident libraries and fonts).

This is a particularly good test in that it can catch problems that might otherwise be missed. Be sure to test from both the CLI and Workbench.

If you have a leak, check that any memory you allocate is deallocated, and anything you open is closed. Particularly insidious are DOS calls, some of which return locks. Three sure clues to a lock problem are:

1) Loss of memory
2) A disk icon that does not "disappear" when the affected disk is removed.
3) A directory or file that returns error 121, "object in use" if an attempt to delete it is made.

$0, $FFFF and Beyond?

Well, that's the end of my list. Maybe you have your own favourite Amiga programming sins you think deserve a little publicity. If so, please share.

Porting from Unix to Amiga

by John Toebes

Amiga–ize a program without changing the original source

John Toebes is a senior systems analyst at SAS Institute Inc. in Cary, North Carolina, and the author of the Lattice C V4.0 compiler. He has been programming the Amiga since 1985, and is the coordinator for The Software Distillery, which is responsible for many popular freely redistributable programs including Hack, PopCLI, MemWatch and Blink. He can be reached on Bix (JTOEBES) and the Software Distillery BBS (919) 471–6436.

With C as the natural system language for the Amiga, there is a wide abundance of programs sitting on Unix and MS-DOS machines begging to be ported. While many of these are utilities such as text formatters and text manipulation routines, there are also a number of games and non-text utilities that would port to the Amiga quite well. However, the rich user interface and multitasking environment on the Amiga provides a different form of user interaction than is available on these other machines. MS-DOS programs typically take over the entire screen and user input, while Unix style programs generally work as filters, reading from one file and writing to another. In both cases, many programs fall back to assuming a character oriented terminal.

This article should familiarize you with common porting problems, and assist in producing programs that, once ported, other Amiga owners will want to use. Many of the techniques presented here come from my experience in porting Hack to the Amiga.

Before attempting to move a program to the Amiga, you should understand that it is not enough to simply recompile the program on the Amiga and say you are done. There are virtually no programs that could not benefit from some "Amigaizing" once the code has been moved. If the program was only to be used once, this would be an acceptable path, but that is rarely the case. If a program was good enough for you to spend time to move over, there is someone else who could benefit just as much. Even more people could benefit from it when it fits well with the Amiga. I have some very simple rules that must apply to a program before I consider it a reasonable Amiga program.

1) It should follow standard Amiga conventions: These include opening standard windows with standard gadgets, supporting the mouse and menus, etc. An "Amigaized" program should understand standard Amiga filenames.
2) It should multitask well: Programs generally don't multitask well when they perform busy-wait loops or provide no means of accessing other programs (for example a full-screen-sized window with no reordering or resizing gadgets).
3) It should use resources carefully: Many programs allocate unnecessarily large areas of memory or open files without closing them later on.
4) It should be pleasing to the eye: This is up to the individual, but some basic guidelines are to keep the display simple, yet use well-designed gadgets.

To understand how to incorporate these principles into a program, let us take a simple RPN-style calculator program written as one would expect to find it on a typical Unix system. Listing 1 (calc.c) shows our program after we have made a few simple changes to start our porting process. These changes, marked by boldface comments in the listing, are simple but essential:

1) Add a #include of "calc.h" so that we have a global header file we can change to reconfigure the way the program works. If your application already has a header file that is included by everything, you don't have to do this. (Listing 2 shows calc.h.)
2) Make sure our program has an explicit call to exit() so that we have a way of catching control when it thinks it is done.
3) Add two new commands to the calculator program that will be useful on the Amiga. We will discuss them further on but present them here so we don't have to change the program later.

Once the program is ready for us to work on, we must first take a moment to understand how it works. We concentrate in particular on how it does its I/O, so that we can determine the best way to take advantage of what the Amiga has to offer. Our calculator is typical of most Unix programs in that it reads the input from the terminal a character at a time until it hits end of file, acting on each character as it comes in. When it wants to say something, it calls printf() to perform the action. We can see this by creating an empty calc.h and then compiling it. If we were using the Lattice C compiler, we would say:

```
echo >calc.h
LC -Lm calc
```

We can run it and then give it some input such as:

```
123 456 + .   ← Our input
579           ← It responds
^\            ← Generate EOF to stop the program
```

The program is usable, but it does not fit the Amiga at all. Our first step is to put the program into its own window and send all input and output through that window. Fortunately for us, the C library routines for reading and writing are not sacred and may be freely replaced. Since our program only uses getchar() and printf(), we only need to replace these functions with our own. To do this, we use the C preprocessor to change all calls to the standard C library routines to ones that we will write. Since we have already created a common include file, this is where we perform the magic:

```
#define getchar mygetc
#define putchar myputc
#define puts myputs
#define printf myprintf
#define exit myexit
```

Listing 2 contains these defines, with the preliminary precaution of undefining the names in case they have been defined previously.

Now, whenever the compiler sees a call to getchar(), it will instead generate a call to mygetc(). We can write these I/O routines as shown in Listing 3 (calcio1.c) to open up a window and then do all I/O through that window. These routines work by using Intuition to create a window (see Chapter 4 of the Intuition Reference Manual for more information), then attaching a "console.device" to it (documented in the RKM Libraries and Devices Manual, pages 275 to 308), which performs all output. Choose a window size that makes sense for the program. In this case we want a small one that could eventually look like a calculator; for other programs (such as Hack or Larn) we are likely to open a window the size of the entire screen.

Input comes through the VANILLAKEY IDCMP interface a character at a time. (See Chapter 8 of the Intuition manual and page 65 of the Enhancer documentation for more about the IDCMP and VANILLAKEY.) If you select the close gadget, an EOF signal is sent to the program. All input is done in the mygetc() function, which handles all messages that come in from Intuition. As a safety measure, if mygetc() ever gets a message that it doesn't understand, it will return an EOF to shut down the program.

Program output is performed through the myputc(), myputs(), and myprintf() routines, which simply construct the appropriate buffer and perform an I/O request with DoIO() to output it. We can now test our program by compiling and then running with:

```
LC -Lm calc calcio
calc
```

We now see that it opens up a separate window into which we can type everything that we did before (although it doesn't echo) *and* we now get immediate output for any command that prints, without having to press RETURN. This is because all input under AmigaDOS is buffered until a carriage return is hit; our input routine returns every character immediately. This has the unfortunate side effect of getting rid of command line editing, but for this program (like many others) it is not a problem and actually turns out to be a benefit.

This program is starting to act as it should on the Amiga, but it certainly doesn't feel like any other Amiga program. The next step is to add a menu (Chapter 6 of the Intuition manual). Although some people may consider menus a frivolous item, they serve as an excellent place for the user to learn what things a program may do without having to read extensive documentation. We can create these menus and add them to our calcio program as shown in Listing 4 (calcio2.c).

This change only requires us to create the menu structures:

MyText holds the text for the items;
MyItems are the descriptions of the items themselves;
MyMenus contain the top level menus.

When creating these by hand, the use of macros such as IDATA makes the final code more readable and readily changed. There are packages on the market that can assist in the task of creating menus (*see the review of Power Windows in this issue – CZ*), but this method works quite well. An important thing to note about our menus is that each item has a character in the CmdKey field specifying the key to be simulated when the menu item is selected.

To actually handle the incoming menu events, we need to add a simple clause to our mygetc() switch statement that looks up the key based on the menu selected. We must also enable menu events for our window in the Window structure, and remember to tell Intuition to enable our menus in mysetup() as well as remove them in myexit(). With these changes in place we can recompile and test with:

```
LC -Lm -M calc calcio
calc
```

. . .and find that our menus now work. This certainly provides immediate visual documentation to the user as to what he can do at any time. The addition of the "?" command in calc and its presence on the menu give us a simple method of telling the user who wrote the program.

Menus are a reasonable start, but we are putting together a calculator. The last time I looked at a calculator, it had buttons on its face to press. Also, it didn't write its answers all over the window – it had a simple display. To make our program appear more like a real calculator, we can add gadgets (Chapter 5 of the Intuition manual) to our window and adjust our output routines as shown in Listing 5 (calcio3.c). We have created the gadgets as we did the menu items with:

GadText to hold the text for the gadget;
MyGads to hold the gadget declarations.

The BUTTON macro allows us to build the gadget structures without having to retype information that is the same for all of them. Like the menu items, we have stored the keystroke to be simulated when the gadget is chosen in the GadgetID field. We chose the X and Y positions for the gadgets in a way that they will look organized on the screen; they could have been placed anywhere and in any order.

To handle the gadgets, we add another clause to the mygetc() routine (is a pattern starting to emerge for adding new input methods?) which extracts the keystroke from the selected gadget. We must also point our window at the first gadget and set the appropriate IDCMP bit to enable gadget messages.

Since we know our calculator only writes a single line at a time, we can modify myprintf() to always position the cursor to a given spot on the screen and then print the string. Since we no longer need the full power of the console to do fancy text output and handle line wraparound, we can drop back to using the simple graphics library Move() and Text() routines to render directly to the window. (These routines are documented in the RKM Libraries and Devices, pages 192 to 194.) Also, myputs() and myputc() are not needed, so we can get rid of them, too. If we expected to do further development of the program and extend its output, we might not elect to remove these routines, but we know what our program does and where it is headed.

Compiling our program and running it brings up a window with a bunch of numbers on the screen that we can click on (or still enter by the keyboard or menu) but it just doesn't have the right look. It is not enough to simply use gadgets – we should make them look like what we are trying to simulate.

In our case we can do a lot by adding borders around the gadgets and adjusting the text in the gadget so that it is centered. Taking a clue from R.J. Mical, we draw not just a box around each gadget, but two boxes to give a nice rounded corner. Using border structures, we create a small border for most of the keys and a large border for the reset key. Pointing the gadget structure to this border and then adjusting the text positions in the GadText array as shown in Listing 6

(calcio4.c), we notice that some of the gadget borders would overlap, so we can fix that at this time. Compiling and running it gives us a reasonable impression of a calculator, but it isn't quite there – the feedback isn't immediate as we would expect. Nothing comes out until we hit "." or "=".

This is where we finally go back to the code we started out with and make a minor change. We add a "p" command, which simply prints out the current number being constructed. (This is already in our program since we knew we would need it). Going back to calcio, we change mygetc() so that for every character it reads in (except for EOF), it sends back a 'p' for the next character. In this way, each time we press a key it will act as if we had also pressed the 'p' key to display the status. This is done by storing 'p' in a static variable at the end of the routine. At the start of the routine, the variable is checked; if it is nonzero, it is set to zero and its value returned. This has the effect of forcing mygetc() to return a 'p' every second time it is called. Now you can see what you're entering into the calculator as it's being entered, instead of working blind and having to press '.' or '=' to see the current result.

In adding this feature, we realize that some commands print something. Since what is printed will get overwritten with the output of the 'p' command, we decide to make myprintf() more intelligent about where it puts the output. When it is printing a number, it will write it at the top; any messages will be written to the bottom. While we are at it, we decide to get rid of the overhead of calling sprintf() and handle the string ourselves, as the program only needs to be able to print decimal and hex numbers. In the sprintf replacement code, the Lattice C functions stci_d and stci_h are used; an #ifdef preprocessor directive skips around this code and uses the standard sprintf function if a compiler other than Lattice is being used.

At this time we can take a final look at our program and clean up just a couple of rough edges. It doesn't use standard I/O or allocate memory, so we can get rid of the compiler overhead by defining our own MemCleanup() routine and adding a #define to calc.h to cause main to appear as _main. We also change all of calcio's calls to _exit to instead call XCEXIT, since we don't need the extra level of cleanup. These common tricks for programs compiled with Lattice C allow us to tell the compiler and linker the level of startup and termination that we want performed by our program. (#ifdef commands in the source will compile the appropriate code for either the Lattice or the Manx Aztec compiler.) Compiling this final program (Listing 7 – calcio.c), we run it to find an RPN calculator that is small, usable *and* fits the Amiga.

What we have done here applies not just to porting a calculator program, but works in a wide variety of applications. The version of calcio.c that we started with is a good basis to start from. It provided the initial interface for Hack, yet allows for successive refinements to improve its interaction with the program and the user. Some additional "Amigaization" steps we can do:

1) Additional output routines that use different fonts (or even graphics). This would require minor changes to the source to call the right routine, but does not dramatically alter the original program.
2) Support for resizing the window and refreshing the display appropriately.
3) Prettier gadgets and menus. The ones here are functional and serve their purpose, but colour and images can be added to them.
4) Support for the mouse on a program such as Hack or a spreadsheet where there are areas on the screen that aren't appropriately served with a gadget.
5) Fancier gadgets such as string and proportional gadgets.

It is important not to be afraid to investigate different ways of doing things. As an example, we switched from using the console.device to the Move and Text routines for this program. Often you will find that one method offers some advantage either in flexibility or in functionality that makes it a better choice.

This process of porting can be summarized into a few steps:

1) Prepare source code with a common header file and explicit exit calls; make sure it compiles first.
2) Understand how the program does its I/O; run it a few times to get a feel for it.
3) Write basic I/O routines; our initial version of calcio is an excellent starting point.
4) Add Amiga features to the I/O routines; use menus and gadgets where appropriate.
5) Add Amiga features to the code; make changes to the code to allow it to better interact with our Amiga I/O routines.

It is wise to start with a simple program before tackling something on the order of Hack, but there are literally hundreds of programs on the Fish disks just begging to be ported properly to the Amiga. One that comes to mind almost immediately is the Chess program on Fish Disk #96. With a little work, someone could easily have a chess board with graphics for the pieces – will you be the first one to do it?

Program listing notes:

Don't be intimidated by the volume of program listings! The entire final version of the calculator program consists of Listings 1, 2 and 7. The others are successive steps taken in the porting process – see the article for details.

This code is for the Lattice compiler, version 4.0 - it should also work with older versions of Lattice. It also compiles with Manx's Aztec C, but you have to use the 32-bit int option (compile with the "+L" flag and link with "-lc32".

Listing 1: The mainline of the RPN calculator program, "calc.c". The only changes that have been made to this program from the original Unix version are indicated by the boldface comments; for the most part, the program remains untouched.

```
#include <stdio.h>
#include <ctype.h>
#include 'calc.h'  /* We added this to make it easier to port */

#define BS 8

int pos, pending;
int stack[50];

int pop()
{
register int rslt;
   if (pos)
       rslt = stack[--pos];
   else
       rslt = 0;
   pending = 0;
   return(rslt);
}

void push(v)
int v;
{
    stack[pos++] = v;
```

```
    pending = 0;
}

void main()
{
register int c;
register int base, v;

    pending = 0;
    pos = 0;
    base = 10;
    stack[pos] = 0;
    while((c = getchar()) != EOF) {
        switch(tolower(c)) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':

            if (pending != 1)
                push(0);
            push((pop() * base) + c - '0');
            pending = 1;
            break;
        case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
            if (base == 16) {
                if (pending != 1)
                    push(0);
                push((pop() * base) + 10 + tolower(c) - 'a');
                pending = 1;
            }
            break;
        case BS:
            if (pending == 1) {
                push(pop() / base);
                pending = 1;
            }
            break;
        case ' ': case '\n':
            pending = 0;
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            v = pop();
            push(pop() - v);
            break;
        case '/':
            v = pop();
            if (v)
                (push(pop() / v));
            else
                printf("Divide by zero\n");
            break;
        case '%':
            v = pop();
            if (v)
                (push(pop() % v));
            else
                printf("Mod by zero\n");
            break;
        case '!':
            push(-pop());
            break;
        case 't':
            base = 10 + ( 6 - (base - 10));
            pending = 0;
            printf("Base is now %d\n", base);
            break;
        case '.':
            v = pop();
            goto displayit;
        case '=':
            push(v = pop());
displayit:
            if (base == 10)
                printf("%d\n", v);
            else
                printf("%x\n", v);
            break;
        case 'r':
            pos = 0;
            stack[pos] = 0;
            pending = 0;
            break;
        case 'q':
            exit(0);
            break;
        /* the question mark displays information about the program */
        case '?':
            printf("Calc by ME!\n");
            pending = 0;
            break;
        /* This case was added so we could simulate the display */
        case 'p':
            v = stack[pos-1];
            goto displayit;
        }
    }
exit(0);
}
```

Listing 2: "calc.h", the header file that redefines the calls to the I/O routines. By including this header file, we can get away with minimum changes to the original source code.

```
#ifdef getchar
#undef getchar
#endif
#define getchar mygetc
#ifdef putchar
#undef putchar
#endif
#define putchar myputc
#define puts myputs
#ifdef printf
#undef printf
#endif
#define printf myprintf
#define exit myexit
#define main _main
```

Listing 3: "calcio1.c", the first version of the file containing the new Amigaized I/O routines that "calc.c" will now use. This version just creates a window for the calculator and lets the user enter input and see results in that window. The IDCMP is used for character input, and the console.device is used for output to the window.

```
#include <dos.h>
#include <intuition/intuition.h>
#include <exec/io.h>

#ifdef LATTICE_V4
/* function prototype headers for Lattice 4.0 */
#include <proto/graphics.h>
#include <proto/intuition.h>
#include <proto/exec.h>
#endif

#ifdef AZTEC_C
#include <functions.h>
#endif

#define EOF -1
#define GRAPHICS_REV  29
#define INTUITION_REV  29

void myexit();

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

static struct IOStdReq conrequest;
struct Window *unixwindow;

/* window to hold calculator */
static struct NewWindow NewWindow = {
    220,25,200,150, -1,-1,  /* left, top, width, height, detail, block */
    CLOSEWINDOW | VANILLAKEY,
    WINDOWDEPTH | WINDOWDRAG | WINDOWCLOSE | ACTIVATE,
    NULL, NULL, (UBYTE *)"My personal calculator",
    NULL, NULL, 200,150,200,150, WBENCHSCREEN };

/* initialize the window so we can read and write to it */
void mysetup()
{
    /* If we haven't already initialized then do so now */
    if (unixwindow == NULL) {
        /* Open up the libraries we will need to use */
        if ( (IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library", INTUITION_REV)) == NULL)
            _exit(1);
```

```
        if ( (GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",GRAPHICS_REV)) == NULL)
            _exit(2);

        /* Open up the window on the workbench screen */
        if ( (unixwindow = (struct Window *)OpenWindow(&NewWindow)) == NULL)
            _exit(3);

        /* And attach a console.device to it for output */
        conrequest.io_Data = (APTR)unixwindow;
        conrequest.io_Length = sizeof(*unixwindow);
        if (OpenDevice("console.device",0,(struct IORequest *)&conrequest, 0))
            myexit(6);
    }
}

int mygetc() /* read a character from our special window */
{
    register int c;
    struct IntuiMessage *Message;

    if (unixwindow == NULL)
        mysetup();   /* Make sure we are initialized */

    /* wait until something happens in the window */
    while( (Message = (struct IntuiMessage *)
                        GetMsg(unixwindow->UserPort)) == NULL)
        Wait( 1 << unixwindow->UserPort->mp_SigBit );

    switch(Message->Class)
    {

      case CLOSEWINDOW:     /* Closing the window certainly should terminate */
                            /* the program with an EOF                       */
                            c = EOF;
                            break;

      case VANILLAKEY:      /* If they gave us the standard EOF character    */
                            /* (^\) Then simulate an EOF to the program      */
                            if ( (c = Message->Code) == 0x1c)
                                c = EOF;
                            /* Translate any carriage returns to a newline   */
                            else if (c == 0x0D)
                                c = '\n';
                            /* Otherwise just a normal character to return   */
                            break;

      default:              /* Any other message we don't recognize clearly */
                            /* is an error so stop immediately to prevent    */
                            /* any possible harm                             */
                            c = EOF;
                            break;
    }
    /* Respond to the message so the system can reuse it */
    ReplyMsg((struct Message *)Message);
    return(c);
}

void myputc(c) /* write a single character to our window */
char c;
{
    if (unixwindow == NULL)
        mysetup();   /* Make sure we are initialized */

    conrequest.io_Command= CMD_WRITE;
    conrequest.io_Data     = (APTR)&c;
    conrequest.io_Length   = 1;
    /* send write request to console.device */
    DoIO((struct IORequest *)&conrequest);
}

void myputs(str) /* write a string to our window */
char *str;
{
    if (unixwindow == NULL)
        mysetup();   /* Make sure we are initialized */

    conrequest.io_Command= CMD_WRITE;
    conrequest.io_Data     = (APTR)str;
    conrequest.io_Length   = strlen(str);
    DoIO((struct IORequest *)&conrequest);
    myputc('\n');    /* don't forget the newline for the string */
}

/* write a formatted string to our window */
void myprintf(str,v1,v2,v3,v4,v5,v6,v7,v8,v9)
char *str;
long v1,v2,v3,v4,v5,v6,v7,v8,v9;
{
    char buff[256];

    if (unixwindow == NULL)
        mysetup();   /* Make sure we are initialized */

    conrequest.io_Length   = sprintf(buff,str,v1,v2,v3,v4,v5,v6,v7,v8,v9);
    conrequest.io_Command= CMD_WRITE;
    conrequest.io_Data     = (APTR)buff;
    DoIO((struct IORequest *)&conrequest);
}

/* close up everything and leave the program */
void myexit(code)
int code;
{
    if (unixwindow != NULL) {
        /* Shut down in the opposite order we came up */
        CloseDevice((struct IORequest *)&conrequest);
        CloseWindow(unixwindow);
        CloseLibrary((struct Library *)GfxBase);
        CloseLibrary((struct Library *)IntuitionBase);
        unixwindow = NULL;
    }
    _exit(code);
}
```

Listing 4: The changes to calcio1.c for our next version, "calcio2.c". This version adds menus for the calculator commands. Structures for the menus and menu items are added, and code to attach and remove the menu strip from the window is added to mysetup() and myexit(). To handle input from the menu, code is added to mygetc(). Only the new structures and changed functions are listed here.

```
/* text for menu items */
static struct IntuiText MyText[] = {
/* Pens mode  Pos   Font    Text                          Next */
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"About        ?",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Toggle Base  t",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Reset        r",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Quit         q",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Add          +",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Subtract     -",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Multiply     *",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Divide       /",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Modulus      %",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Negate       !",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Print        .",   NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)"Print Top    =",   NULL } };

#define MENUFLAGS ITEMTEXT | ITEMENABLED | HIGHCOMP
#define IDATA(off,str,key) 0,off,110,9,MENUFLAGS,0,str,NULL,key,NULL,NULL

/* menu items */
struct MenuItem MyItems[] = {
#define LIST1 0
 { &MyItems[1], IDATA( 0,  (APTR)&MyText[0],  '?')  },
 { &MyItems[2], IDATA(10, (APTR)&MyText[1],  't')  },
 { &MyItems[3], IDATA(20, (APTR)&MyText[2],  'r')  },
 { NULL,        IDATA(30, (APTR)&MyText[3],  'q')  },
#define LIST2 4
 { &MyItems[5], IDATA( 0,  (APTR)&MyText[4],  '+')  },
 { &MyItems[6], IDATA(10, (APTR)&MyText[5],  '-')  },
 { &MyItems[7], IDATA(20, (APTR)&MyText[6],  '*')  },
 { &MyItems[8], IDATA(30, (APTR)&MyText[7],  '/')  },
 { &MyItems[9], IDATA(40, (APTR)&MyText[8],  '%')  },
 { &MyItems[10],IDATA(50,(APTR)&MyText[9],  '!')  },
 { &MyItems[11],IDATA(60,(APTR)&MyText[10], '.')  },
 { NULL,        IDATA(70, (APTR)&MyText[11], '=')  } };

/* menus */
static struct Menu MyMenus[] = {
 { NULL,          0,    0, 100, 10, MENUENABLED,
        (BYTE *)"Project",      &MyItems[LIST1] },
 { &MyMenus[0], 101,  0, 100, 10, MENUENABLED,
        (BYTE *)"Commands", &MyItems[LIST2] } };
#define LASTMENU 1

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

static struct IOStdReq conrequest;
struct Window *unixwindow;

/* window to hold calculator */
static struct NewWindow NewWindow = {
    220,25,200,150, -1,-1,  /* left, top, width, height, detail, block */
    MENUPICK | CLOSEWINDOW | VANILLAKEY,
    WINDOWDEPTH | WINDOWDRAG | WINDOWCLOSE | ACTIVATE,
    NULL, NULL, (UBYTE *)"My personal calculator",
    NULL, NULL, 200,150,200,150, WBENCHSCREEN };

/* initialize the window so we can read and write to it */
```

```
void mysetup()
{
    /* If we haven't already initialized then do so now */
    if (unixwindow == NULL) {
        /* Open up the libraries we will need to use */
        if ( (IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library", INTUITION_REV)) == NULL)
            _exit(1);

        if ( (GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",GRAPHICS_REV)) == NULL)
            _exit(2);

        /* Open up the window on the workbench screen */
        if ( (unixwindow = (struct Window *)OpenWindow(&NewWindow)) == NULL)
            _exit(3);

        /* And attach a console.device to it for output */
        conrequest.io_Data     = (APTR)unixwindow;
        conrequest.io_Length = sizeof(*unixwindow);
        if (OpenDevice("console.device",0,(struct IORequest *)&conrequest, 0))
            myexit(6);

        /* Tell Intuition about our menu */
        SetMenuStrip(unixwindow, &MyMenus[LASTMENU]);
    }
}

int mygetc() /* read a character from our special window */
{
    register int c;
    long MenuNumber;
    struct MenuItem *item;
    register struct IntuiMessage *Message;

    if (unixwindow == NULL)
        mysetup();   /* Make sure we are initialized */

    /* wait until something happens in the window */
    while( (Message = (struct IntuiMessage *)
                        GetMsg(unixwindow->UserPort)) == NULL)
        Wait( 1 << unixwindow->UserPort->mp_SigBit );

    switch(Message->Class)
    {

      case CLOSEWINDOW:     /* Closing the window certainly should terminate */
                            /* the program with an EOF                       */
                            c = EOF;
                            break;

      case VANILLAKEY:      /* If they gave us the standard EOF character    */
                            /* (^\) Then simulate an EOF to the program      */
                            if ( (c = Message->Code) == 0x1c)
                                c = EOF;
                            /* Translate any carriage returns to a newline   */
                            else if (c == 0x0D)
                                c = '\n';
                            /* Otherwise just a normal character to return   */
                            break;

      case MENUPICK:        /* When a menu comes in, we need to chase the    */
                            /* list of menu items selected and use just the  */
                            /* last one they asked for                       */
                            MenuNumber = Message->Code;
                            while(MenuNumber != MENUNULL) {
                                item = (struct MenuItem*)
                                        ItemAddress(&MyMenus[LASTMENU],
                                        MenuNumber);
                                c = item->Command;
                                MenuNumber = item->NextSelect;
                            }
                            break;

      default:              /* Any other message we don't recognize clearly */
                            /* is an error so stop immediately to prevent   */
                            /* any possible harm                            */
                            c = EOF;
                            break;
    }
    /* Respond to the message so the system can reuse it */
    ReplyMsg((struct Message *)Message);
    return(c);
}

/* the code in here remains the same as "calcio1.c" */

/* close up everything and leave the program */

void myexit(code)
int code;
{
    if (unixwindow != NULL) {
        /* Shut down in the opposite order we came up */
        CloseDevice((struct IORequest *)&conrequest);
        ClearMenuStrip(unixwindow);
        CloseWindow(unixwindow);
        CloseLibrary((struct Library *)GfxBase);
        CloseLibrary((struct Library *)IntuitionBase);
        unixwindow = NULL;
    }
    _exit(code);
}
```

Listing 5: The additions and changes to move to the next phase of porting, "calcio3.c". In this version, gadgets are added for the calculator keys and the calculator output display is improved. IntuiText structures for the gadgets and the Gadget structures themselves are added. The functions replaced are mysetup(), mygetc(), myprintf(), and myexit(). The functions myputc() and myputs() are removed. Note that the menu-related structures from the previous listing (everything before the functions start) remain the same for this version.

```
/* text for Gadgets */
static struct IntuiText GadText[] = {
/* Pens mode  Pos    Font   Text                    Next */
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"0",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"1",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"2",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"3",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"4",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"5",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"6",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"7",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"8",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"9",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"A",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"B",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"C",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"D",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"E",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"F",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"*",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"/",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"+",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"-",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"%",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"=",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)".",          NULL },
 { 1, 0,  JAM1, 0, 0,  NULL, (UBYTE *)"RESET",      NULL } };

#define BUTTON(num,c) GADGHCOMP,RELVERIFY,BOOLGADGET,NULL,NULL,\
                                &GadText[num],0,NULL,c,NULL

/* Gadgets for calculator's buttons */
static struct Gadget MyGads[] = {
 /* Next gad      X   Y    Wdth Hgt   Text#      Key */
 { &MyGads[1],   10, 30,   12,   12, BUTTON(0,   '0' ) },
 { &MyGads[2],   30, 30,   12,   12, BUTTON(1,   '1' ) },
 { &MyGads[3],   50, 30,   12,   12, BUTTON(2,   '2' ) },
 { &MyGads[4],   70, 30,   12,   12, BUTTON(3,   '3' ) },
 { &MyGads[5],   10, 50,   12,   12, BUTTON(4,   '4' ) },
 { &MyGads[6],   30, 50,   12,   12, BUTTON(5,   '5' ) },
 { &MyGads[7],   50, 50,   12,   12, BUTTON(6,   '6' ) },
 { &MyGads[8],   70, 50,   12,   12, BUTTON(7,   '7' ) },
 { &MyGads[9],   10, 70,   12,   12, BUTTON(8,   '8' ) },
 { &MyGads[10],  30, 70,   12,   12, BUTTON(9,   '9' ) },
 { &MyGads[11],  50, 70,   12,   12, BUTTON(10,  'A' ) },
 { &MyGads[12],  70, 70,   12,   12, BUTTON(11,  'B' ) },
 { &MyGads[13],  10, 90,   12,   12, BUTTON(12,  'C' ) },
 { &MyGads[14],  30, 90,   12,   12, BUTTON(13,  'D' ) },
 { &MyGads[15],  50, 90,   12,   12, BUTTON(14,  'E' ) },
 { &MyGads[16],  70, 90,   12,   12, BUTTON(15,  'F' ) },
 { &MyGads[17], 100, 30,   12,   12, BUTTON(16,  '*' ) },
 { &MyGads[18], 100, 50,   12,   12, BUTTON(17,  '/' ) },
 { &MyGads[19], 100, 70,   12,   12, BUTTON(18,  '+' ) },
 { &MyGads[20], 100, 90,   12,   12, BUTTON(19,  '-' ) },
 { &MyGads[21], 100, 110,  12,   12, BUTTON(20,  '%' ) },
 { &MyGads[22],  10, 110,  12,   12, BUTTON(21,  '=' ) },
 { &MyGads[23],  30, 110,  12,   12, BUTTON(22,  '.' ) },
 { NULL,         50, 110,  48,   12, BUTTON(23,  'r' ) } };

/* menu structures and other code from previous listings go here */

/* window to hold calculator */
static struct NewWindow NewWindow = {
    220,25,200,150, -1,-1,  /* left, top, width, height, detail, block */
    MENUPICK | GADGETUP | GADGETDOWN | CLOSEWINDOW | VANILLAKEY,
```

```
    WINDOWDEPTH | WINDOWDRAG | WINDOWCLOSE | ACTIVATE,
    &MyGads[0], NULL, (UBYTE *)"My personal calculator",
    NULL, NULL, 200,150,200,150, WBENCHSCREEN };

/* initialize the window so we can read and write to it */
void mysetup()
{
    /* If we haven't already initialized then do so now */
    if (unixwindow == NULL)
    {
        /* Open up the libraries we will need to use */
        if ( (IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library", INTUITION_REV)) == NULL)
            _exit(1);

        if ( (GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",GRAPHICS_REV)) == NULL)
            _exit(2);

        /* Open up the window on the workbench screen */
        if ( (unixwindow = (struct Window *)OpenWindow(&NewWindow)) == NULL)
            _exit(3);

        /* Tell Intuition about our menu */
        SetMenuStrip(unixwindow, &MyMenus[LASTMENU]);
    }
}

int mygetc() /* read a character from our special window */
{
    register int c;
    long MenuNumber;
    struct Gadget *gad;
    struct MenuItem *item;
    register struct IntuiMessage *Message;

    if (unixwindow == NULL)
        mysetup();   /* Make sure we are initialized */

    /* wait until something happens in the window */
    while( (Message = (struct IntuiMessage *)
                        GetMsg(unixwindow->UserPort)) == NULL)
        Wait( 1 << unixwindow->UserPort->mp_SigBit );

    switch(Message->Class)
    {

      case CLOSEWINDOW:     /* Closing the window certainly should terminate */
                            /* the program with an EOF */
                            c = EOF;
                            break;

      case VANILLAKEY:      /* If they gave us the standard EOF character    */
                            /* (^\) Then simulate an EOF to the program      */
                            if ( (c = Message->Code) == 0x1C)
                                c = EOF;
                            /* Translate any carriage returns to a newline   */
                            else if (c == 0x0D)
                                c = '\n';
                            /* Otherwise just a normal character to return   */
                            break;

      case GADGETDOWN:
      case GADGETUP:        /* They selected a gadget - use id as the key    */
                            gad = (struct Gadget *)Message->IAddress;
                            c = gad->GadgetID;
                            break;

      case MENUPICK:        /* When a menu comes in, we need to chase the */
                            /* list of menu items selected and use just the  */
                            /* last one they asked for                       */
                            MenuNumber = Message->Code;
                            while(MenuNumber != MENUNULL) {
                                item = (struct MenuItem*)
                                        ItemAddress(&MyMenus[LASTMENU],
                                        MenuNumber);
                                c = item->Command;
                                MenuNumber = item->NextSelect;
                            }
                            break;

      default:              /* Any other message we don't recognize clearly */
                            /* is an error so stop immediately to prevent    */
                            /* any possible harm                             */
                            c = EOF;
                            break;
    }
    /* Respond to the message so the system can reuse it */
    ReplyMsg((struct Message *)Message);
    return(c);
}

/* write a formatted string to our window */
void myprintf(str,v1,v2,v3,v4,v5,v6,v7,v8,v9)
char *str;
long v1,v2,v3,v4,v5,v6,v7,v8,v9;
{
    char buff[256];
    int i;

    if (unixwindow == NULL)
        mysetup();   /* Make sure we are initialized */

    i = sprintf(buff,str,v1,v2,v3,v4,v5,v6,v7,v8,v9);
    Move(unixwindow->RPort, 4, 20);
    /* eighteen spaces between quotes below */
    Text(unixwindow->RPort, "[18 spcs]", 18);
    Move(unixwindow->RPort, 4, 20);
    Text(unixwindow->RPort, buff, i-1);
}

/* close up everything and leave the program */
void myexit(code)
int code;
{
    if (unixwindow != NULL) {
        /* Shut down in the opposite order we came up */
        ClearMenuStrip(unixwindow);
        CloseWindow(unixwindow);
        CloseLibrary((struct Library *)GfxBase);
        CloseLibrary((struct Library *)IntuitionBase);
        unixwindow = NULL;
    }
    _exit(code);
}
```

Listing 6: Changes for the next version, "calcio4.c". This version adds borders around the gadgets to make them look like calculator buttons. The border end-point arrays and border structures are added, and the gadget text and structures are replaced. The rest of the code is the same as the previous listings.

```
/* border end points */
SHORT smallbox[] =
{ -2, -1,  13, -1,  13, 12,  -2, 12,  -2, -1,
  -1, -2,  12, -2,  12, 13,  -1, 13,  -1, -2 };
#define SVECTORS (sizeof(smallbox)/(2*sizeof(short)))

SHORT largebox[] =
{ -2, -1,  49, -1,  49, 12,  -2, 12,  -2, -1,
  -1, -2,  48, -2,  48, 13,  -1, 13,  -1, -2 };
#define LVECTORS (sizeof(largebox)/(2*sizeof(short)))

/* borders for button Gadgets */
struct Border SmallBorder =
        {0,0,3,2,JAM1,SVECTORS,(SHORT*)&smallbox,  NULL};
struct Border LargeBorder =
        {0,0,3,2,JAM1,LVECTORS,(SHORT*)&largebox,  NULL};

/* text for Gadgets */
static struct IntuiText GadText[] = {
/* Pens mode  Pos   Font   Text              Next */
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"0",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"1",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"2",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"3",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"4",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"5",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"6",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"7",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"8",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"9",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"A",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"B",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"C",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"D",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"E",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"F",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"*",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"/",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"+",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"-",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"%",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"=",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)".",  NULL },
  { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)"RESET", NULL } };

#define BUTTON(num,box,c) GADGHCOMP,RELVERIFY,BOOLGADGET,box,NULL,\
                                    &GadText[num],0,NULL,c,NULL
```

```
/* Gadgets for calculator's buttons */
static struct Gadget MyGads[] = {
 /* Next gad      X   Y   Wdth Hgt   Text#      Border                  Key */
 { &MyGads[1],   10, 30, 12,  12, BUTTON(0, (APTR)&SmallBorder,  '0')  },
 { &MyGads[2],   30, 30, 12,  12, BUTTON(1, (APTR)&SmallBorder,  '1')  },
 { &MyGads[3],   50, 30, 12,  12, BUTTON(2, (APTR)&SmallBorder,  '2')  },
 { &MyGads[4],   70, 30, 12,  12, BUTTON(3, (APTR)&SmallBorder,  '3')  },
 { &MyGads[5],   10, 50, 12,  12, BUTTON(4, (APTR)&SmallBorder,  '4')  },
 { &MyGads[6],   30, 50, 12,  12, BUTTON(5, (APTR)&SmallBorder,  '5')  },
 { &MyGads[7],   50, 50, 12,  12, BUTTON(6, (APTR)&SmallBorder,  '6')  },
 { &MyGads[8],   70, 50, 12,  12, BUTTON(7, (APTR)&SmallBorder,  '7')  },
 { &MyGads[9],   10, 70, 12,  12, BUTTON(8, (APTR)&SmallBorder,  '8')  },
 { &MyGads[10],  30, 70, 12,  12, BUTTON(9, (APTR)&SmallBorder,  '9')  },
 { &MyGads[11],  50, 70, 12,  12, BUTTON(10, (APTR)&SmallBorder, 'A')  },
 { &MyGads[12],  70, 70, 12,  12, BUTTON(11, (APTR)&SmallBorder, 'B')  },
 { &MyGads[13],  10, 90, 12,  12, BUTTON(12, (APTR)&SmallBorder, 'C')  },
 { &MyGads[14],  30, 90, 12,  12, BUTTON(13, (APTR)&SmallBorder, 'D')  },
 { &MyGads[15],  50, 90, 12,  12, BUTTON(14, (APTR)&SmallBorder, 'E')  },
 { &MyGads[16],  70, 90, 12,  12, BUTTON(15, (APTR)&SmallBorder, 'F')  },
 { &MyGads[17], 110, 30, 12,  12, BUTTON(16, (APTR)&SmallBorder, '*')  },
 { &MyGads[18], 110, 50, 12,  12, BUTTON(17, (APTR)&SmallBorder, '/')  },
 { &MyGads[19], 110, 70, 12,  12, BUTTON(18, (APTR)&SmallBorder, '+')  },
 { &MyGads[20], 110, 90, 12,  12, BUTTON(19, (APTR)&SmallBorder, '-')  },
 { &MyGads[21], 110, 110, 12, 12, BUTTON(20, (APTR)&SmallBorder, '%')  },
 { &MyGads[22],  10, 110, 12, 12, BUTTON(21, (APTR)&SmallBorder, '=')  },
 { &MyGads[23],  30, 110, 12, 12, BUTTON(22, (APTR)&SmallBorder, '.')  },
 { NULL,         50, 110, 48, 12, BUTTON(23, (APTR)&LargeBorder, 'r')  } };
```

Listing 7: The final version of "calcio.c", in its entirety; this is the only version you need to type in. The display has been greatly improved from the previous version, because it is continuously updated as keys are entered. The myprintf() function has been rewritten to eliminate the call to sprintf(), shortening the program. A few other size optimizations have been made, including changing the calls to _exit() with XCEXIT().

```
#include <dos.h>
#include <intuition/intuition.h>
#include <exec/io.h>

#ifdef LATTICE_V4
/* function prototype headers for Lattice 4.0 */
#include <proto/graphics.h>
#include <proto/intuition.h>
#include <proto/exec.h>
#endif

#ifdef AZTEC_C
#include <functions.h>
#define XCEXIT(a) _exit(a)
#endif

#define EOF -1
#define GRAPHICS_REV  29
#define INTUITION_REV 29

/* border end points */
SHORT smallbox[] =
{ -2, -1,  13, -1,  13, 12,  -2, 12,  -2, -1,
  -1, -2,  12, -2,  12, 13,  -1, 13,  -1, -2 };
#define SVECTORS (sizeof(smallbox)/(2*sizeof(short)))

SHORT largebox[] =
{ -2, -1,  49, -1,  49, 12,  -2, 12,  -2, -1,
  -1, -2,  48, -2,  48, 13,  -1, 13,  -1, -2 };
#define LVECTORS (sizeof(largebox)/(2*sizeof(short)))

/* borders for button Gadgets */
struct Border SmallBorder =
        { 0,0,3,2,JAM1, SVECTORS,(SHORT*)&smallbox,  NULL};
struct Border LargeBorder =
        { 0,0,3,2,JAM1, LVECTORS,(SHORT*)&largebox,  NULL};

/* text for Gadgets */
static struct IntuiText GadText[] = {
/* Pens mode  Pos   Font   Text                  Next */
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'0',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'1',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'2',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'3',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'4',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'5',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'6',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'7',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'8',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'9',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'A',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'B',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'C',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'D',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'E',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'F',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'*',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'/',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'+',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'-',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'%',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'=',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)'.',  NULL },
 { 1, 0,  JAM1, 2, 2,  NULL, (UBYTE *)' RESET', NULL } };

#define BUTTON(num,box,c) GADGHCOMP,RELVERIFY,BOOLGADGET,box,NULL,\
                                &GadText[num],0,NULL,c,NULL

/* Gadgets for calculator's buttons */
static struct Gadget MyGads[] = {
 /* Next gad      X   Y   Wdth Hgt   Text#      Border                  Key */
 { &MyGads[1],   10, 30, 12,  12, BUTTON(0, (APTR)&SmallBorder,  '0')  },
 { &MyGads[2],   30, 30, 12,  12, BUTTON(1, (APTR)&SmallBorder,  '1')  },
 { &MyGads[3],   50, 30, 12,  12, BUTTON(2, (APTR)&SmallBorder,  '2')  },
 { &MyGads[4],   70, 30, 12,  12, BUTTON(3, (APTR)&SmallBorder,  '3')  },
 { &MyGads[5],   10, 50, 12,  12, BUTTON(4, (APTR)&SmallBorder,  '4')  },
 { &MyGads[6],   30, 50, 12,  12, BUTTON(5, (APTR)&SmallBorder,  '5')  },
 { &MyGads[7],   50, 50, 12,  12, BUTTON(6, (APTR)&SmallBorder,  '6')  },
 { &MyGads[8],   70, 50, 12,  12, BUTTON(7, (APTR)&SmallBorder,  '7')  },
 { &MyGads[9],   10, 70, 12,  12, BUTTON(8, (APTR)&SmallBorder,  '8')  },
 { &MyGads[10],  30, 70, 12,  12, BUTTON(9, (APTR)&SmallBorder,  '9')  },
 { &MyGads[11],  50, 70, 12,  12, BUTTON(10, (APTR)&SmallBorder, 'A')  },
 { &MyGads[12],  70, 70, 12,  12, BUTTON(11, (APTR)&SmallBorder, 'B')  },
 { &MyGads[13],  10, 90, 12,  12, BUTTON(12, (APTR)&SmallBorder, 'C')  },
 { &MyGads[14],  30, 90, 12,  12, BUTTON(13, (APTR)&SmallBorder, 'D')  },
 { &MyGads[15],  50, 90, 12,  12, BUTTON(14, (APTR)&SmallBorder, 'E')  },
 { &MyGads[16],  70, 90, 12,  12, BUTTON(15, (APTR)&SmallBorder, 'F')  },
 { &MyGads[17], 110, 30, 12,  12, BUTTON(16, (APTR)&SmallBorder, '*')  },
 { &MyGads[18], 110, 50, 12,  12, BUTTON(17, (APTR)&SmallBorder, '/')  },
 { &MyGads[19], 110, 70, 12,  12, BUTTON(18, (APTR)&SmallBorder, '+')  },
 { &MyGads[20], 110, 90, 12,  12, BUTTON(19, (APTR)&SmallBorder, '-')  },
 { &MyGads[21], 110, 110, 12, 12, BUTTON(20, (APTR)&SmallBorder, '%')  },
 { &MyGads[22],  10, 110, 12, 12, BUTTON(21, (APTR)&SmallBorder, '=')  },
 { &MyGads[23],  30, 110, 12, 12, BUTTON(22, (APTR)&SmallBorder, '.')  },
 { NULL,         50, 110, 48, 12, BUTTON(23, (APTR)&LargeBorder, 'r')  } };

/* text for menu items */
static struct IntuiText MyText[] = {
/* Pens mode  Pos   Font   Text                      Next */
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'About        ?',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Toggle Base  t',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Reset        r',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Quit         q',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Add          +',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Subtract     -',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Multiply     *',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Divide       /',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Modulus      %',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Negate       !',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Print        .',  NULL },
 { 0, 1,  JAM1, 0, 0,  NULL, (UBYTE *)'Print Top    =',  NULL } };

#define MENUFLAGS ITEMTEXT | ITEMENABLED | HIGHCOMP
#define IDATA(off,str,key) 0,off,110,9,MENUFLAGS,0,str,NULL,key,NULL,NULL

/* menu items */
struct MenuItem MyItems[] = {
#define LIST1 0
 { &MyItems[1],  IDATA( 0, (APTR)&MyText[0], '?')  },
 { &MyItems[2],  IDATA(10, (APTR)&MyText[1], 't')  },
 { &MyItems[3],  IDATA(20, (APTR)&MyText[2], 'r')  },
 { NULL,         IDATA(30, (APTR)&MyText[3], 'q')  },
#define LIST2 4
 { &MyItems[5],  IDATA( 0, (APTR)&MyText[4], '+')  },
 { &MyItems[6],  IDATA(10, (APTR)&MyText[5], '-')  },
 { &MyItems[7],  IDATA(20, (APTR)&MyText[6], '*')  },
 { &MyItems[8],  IDATA(30, (APTR)&MyText[7], '/')  },
 { &MyItems[9],  IDATA(40, (APTR)&MyText[8], '%')  },
 { &MyItems[10], IDATA(50, (APTR)&MyText[9], '!')  },
 { &MyItems[11], IDATA(60, (APTR)&MyText[10], '.')  },
 { NULL,         IDATA(70, (APTR)&MyText[11], '=')  } };

/* menus */
static struct Menu MyMenus[] = {
 { NULL,          0,  0, 100, 10, MENUENABLED,
 "Project",                &MyItems[LIST1] },
 { &MyMenus[0], 101,  0, 100, 10, MENUENABLED,
 "Commands",               &MyItems[LIST2] } };
#define LASTMENU 1
```

```
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

struct Window *unixwindow;

/* window to hold calculator */
static struct NewWindow NewWindow = {
    220,25,200,150, -1,-1,  /* left, top, width, height, detail, block */
    MENUPICK | GADGETUP | GADGETDOWN | CLOSEWINDOW | VANILLAKEY,
    WINDOWDEPTH | WINDOWDRAG | WINDOWCLOSE | ACTIVATE,
    &MyGads[0], NULL, (UBYTE *)"MyCalc V0.0",
    NULL, NULL, 200,150,200,150, WBENCHSCREEN };

/* initialize the window so we can read and write to it */
void mysetup()
{
    /* If we haven't already initialized then do so now */
    if (unixwindow == NULL) {
        /* Open up the libraries we will need to use */
        if ( (IntuitionBase = (struct IntuitionBase *)
            OpenLibrary("intuition.library", INTUITION_REV)) == NULL)
            XCEXIT(1);

        if ( (GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library",GRAPHICS_REV)) == NULL)
            XCEXIT(2);

        /* Open up the window on the workbench screen */
        if ( (unixwindow = (struct Window *)OpenWindow(&NewWindow)) == NULL)
            XCEXIT(3);

        /* Tell Intuition about our menu */
        SetMenuStrip(unixwindow, &MyMenus[LASTMENU]);
    }
}

int mygetc() /* read a character from our special window */
{
register int c;
long MenuNumber;
struct MenuItem *item;
struct Gadget *gad;
register struct IntuiMessage *Message;
static int savec = 0;

    if (unixwindow == NULL)
        mysetup(); /* Make sure we have initialized */

    /* If we queued up a character then return it now */
    if (c = savec) {
        savec = 0;
        return(c);
    }
    /* wait until something happens in the window */
    while( (Message = (struct IntuiMessage *)
                        GetMsg(unixwindow->UserPort)) == NULL)
        Wait( 1 << unixwindow->UserPort->mp_SigBit );

    switch(Message->Class)
    {
      case CLOSEWINDOW:     /* Closing the window certainly should terminate */
                            /* the program with an EOF                        */
                            c = EOF;
                            break;

      case VANILLAKEY:      /* If they gave us the standard EOF character     */
                            /* (^\) Then simulate an EOF to the program       */
                            if ( (c = Message->Code) == 0x1c)
                                c = EOF;
                            /* Translate any carriage returns to a newline    */
                            else if (c == 0x0D)
                                c = '\n';
                            /* Otherwise just a normal character to return    */
                            break;

      case GADGETDOWN:
      case GADGETUP:        /* They selected a gadget - use ID as the key     */
                            gad = (struct Gadget *)Message->IAddress;
                            c = gad->GadgetID;
                            break;

      case MENUPICK:        /* When a menu comes in, we need to chase the     */
                            /* list of menu items selected and use just the   */
                            /* last one they asked for                        */
                            MenuNumber = Message->Code;
                            while(MenuNumber != MENUNULL) {
                                item = (struct MenuItem*)
                                        ItemAddress(&MyMenus[LASTMENU],
                                        MenuNumber);
                                c = item->Command;
                                MenuNumber = item->NextSelect;
                            }
                            break;

      default:              /* Any other message we don't recognize clearly */
                            /* is an error so stop immediately to prevent   */
                            /* any possible harm                            */
                            c = EOF;
                            break;
    }
    /* Respond to the message so the system can reuse it */
    ReplyMsg((struct Message *)Message);

    /* See if we have to queue up a character for next time */
    if (c != EOF)
        savec = 'p';
    return(c);
}

/* write a formatted string to our window in a particular location */
void myprintf(str,v)
register char *str;
long v;
{
    char buff[256];
    register char *p;
    int line, linelen;

    if (unixwindow == NULL)
        mysetup();

    /* Determine if this is a message or a print of a value */
    if ((*str == '%') || (*str == '$'))
        line = 20;  /* Values go on the top */
    else
        line = 140; /* messages go on the bottom */

#ifdef LATTICE /* use Lattice's stci_d and stci_h to avoid using sprintf */
    p = buff;
    while(*str) {
        /* Is this character a formatting code */
        if (*str == '%') {
            str++;
            if (*str == 'd') {
                /* Decimal Conversion */
                p += stci_d(p, v);
                str++;
            }
            else if (*str == 'x') {
                /* Hex conversion */
                *p++ = '$';
                p += stci_h(p, v);
                str++;
            }
            else /* Normal character, just copy it over */
                *p++ = *str++;
        }
        else if (*str != '\n')
            *p++ = *str++; /* except for newlines */
        else
            str++;      /* Ignore the newline  */
    }
    linelen = p - buff;
#else /* if not using Lattice, use sprintf and live with the extra code */
    linelen = sprintf(buff, str, v) - 1;
#endif

    /* Blank out the line first */
    Move(unixwindow->RPort, 4, line);
    /* eighteen spaces between the quotes below */
    Text(unixwindow->RPort, "[18 spcs]", 18);
    /* and then print the message */
    Move(unixwindow->RPort, 4, line);
    Text(unixwindow->RPort, buff, linelen);
}

/* close up everything and leave the program */
void myexit(code)
int code;
{
    if (unixwindow != NULL) {
        /* Shut down in the opposite order we came up */
        ClearMenuStrip(unixwindow);
        CloseWindow(unixwindow);
        CloseLibrary((struct Library *)GfxBase);
        CloseLibrary((struct Library *)IntuitionBase);
        unixwindow = NULL;
    }
    XCEXIT(code);
}

void MemCleanup(){}
```

Entering New Dimensions
Memory Allocation and Lists: An Introduction

by Rob Peck

Using Exec Lists helps give your programs room to grow!

Rob Peck is the author of the Amiga ROM Kernel and Amiga Hardware Manuals, and the Programmers' Guide To The Amiga (SYBEX, 1987). He can be reached through: DATAPATH, P.O. Box 1828, Los Gatos, CA, 95031-1828.

Ever create a program that was supposed to handle a specific amount of data, only to find that you eventually wanted it to cope with more items than you first anticipated? This usually means you'll have to go back into the program and increase the size of the arrays you're using to store the data. Instead of doing this, though, you might consider using dynamically created data items, allowing your program to handle whatever size of arrays you'll use. As long as there is enough memory to handle everything, the program needn't be changed.

This article assumes that you are familiar with programming in BASIC, and that you perhaps have seen C language programs for the Amiga in books and magazines. Unless you have studied the C language, those C listings will probably be difficult for you to understand. To introduce the concepts of dynamic data storage, I've begun with an explanation of data structures in C, then I've shown how data structures are dynamically allocated, and finally I've shown how you can construct lists of data structures and thereby keep track of things.

What You Do In BASIC

Let's start out by thinking about how people often handle data when programming in BASIC. Let's say you want to create a collection of data about people, perhaps a mailing list. If you have 100 people on that list, you might create a program that has the following variables defined. I have included them in DIM statements for convenience:

```
DIM NAME$(100), ADDRESS$(100)
DIM CITY$(100), STATE$(100), ZIP(100)
```

To work with this data in your program, you would choose an INDEX variable and use it to refer to one of each of the items within the various arrays, assigning values to the items in each array as you went along. Here is an example:

```
INDEX = INDEX + 1

NAME$(INDEX) = "Neighbor Sam"
ADDRESS$(INDEX) = "12345 Harbor Lane"
```

and so on.

Why Look For Another Way?

Before I learned Pascal and C, I thought that this was the way everybody organized groups of data – in indexed sets of arrays. That's the way it seemed to be in FORTRAN, and I carried it over to BASIC as well. Why would anyone want to do it any other way?

Well, I learned that there are a lot of reasons one might want to do it other ways. Organizing the data into blocks of directly related material, instead of several disjoint arrays, makes the data easier to handle. . . easier to think about as an entity. If you have to handle groups of data, it is just as easy to work with arrays of those entities (each item in the array being made up of several smaller items, each at a known position within the group).

Stuck With A Fixed Number Of Elements

The DIM statements above impose a fixed limit on the amount of data the program can handle. If I have already created 100 items, and I now need to add another one, I have to modify the program to allow for more items.

Some programming languages allow you to create items in a dynamic manner. The language asks the operating system to reserve a set of memory locations large enough to contain the complete definition of an item, then tells us where this memory can be found. We can add another item to a group that we are forming by copying the data into this newly-allocated memory.

That is exactly what we'll be leading up to here – using the Amiga's memory allocation facilities to provide space for each new item as it is created. We'll create a data structure; that is, we'll select a name and use it to stand for the way that a

particular set of data is organized. Then we will look at how the operating system can be asked to give us some memory to work with. Finally we'll add this new item to a list of items that we can manipulate any way we wish.

We'll start by defining a data structure that is the rough equivalent of what you see in the BASIC example above. Rather than try to simulate data structures in BASIC, let's go directly to C.

Note that anything between the character pairs "/*" and "*/" is a comment that is entirely ignored by the C compiler. Comments are just like the REM statement in BASIC, but may span multiple lines if the programmer is so inclined. (A side note: when I program, I often install so many comments that people complain that they find it difficult to see the code itself – sigh. . . can't please everyone.)

```
/* assume this is in a file named "label.h" */

struct Label {
   struct Node Lnode; /* for linking things */
                      /* occupies 14 bytes */
   char name[30];     /* reserve 30 for the name$ */
   char address[50];  /* and 50 for the address$ */
   char city[20];     /* and 20 for city */
   char state[2];     /* and  2 for state */
   LONG zip;          /* 4 bytes for zipcode integer value */
};
```

What you see here is called a "declaration" of the Label data structure. A data structure is an organization of data to which you can give a single name. There are many different data structures that you can build, and it really depends on the application you are trying to perform exactly how many data structures you might want to use and how you might want to organize them.

The declaration above begins with the notation:

```
struct Label {
```

which means, to the C language, that I am defining a single data structure that is to be referred to by the name "Label". The curly brackets indicate the starting and ending points of a specific grouping of items.

In many cases, when you see data structures defined (such as in the Amiga include files – C compiler files whose names end in ".h") you'll see data structures defined like this:

```
struct item {
   /* some stuff here */
};
```

When you see this, it means that the file is only *defining* the data structure, and the C compiler is not supposed to reserve any memory locations for it in your program unless you actually use a data element of that type. Such a file is rather like a dictionary in which the compiler will look up the definition of a data structure when (and if) that definition is needed.

Sometimes you'll see declarations take this form instead:

```
struct item another_item = {
   /* some stuff here */
};
```

Notice the equals (=) sign in this other kind of declaration. When the equals is present, it tells the compiler this is *not* a definition, but is rather an entity for which the compiler is supposed to reserve memory for storing data.

The second line in the "Label" data structure defined above is another data structure, called a Node. (C lets you describe data structures inside of other data structures.) We'll skip over this for the moment. It has to do with forming lists of data items, which is a subject we'll return to later.

The next few items in the Label declaration are:

```
char name[30];
char address[50];
and so on. . .
```

A "char" declaration tells the compiler to reserve memory spaces for storage of characters (such as letters and numbers). So a char[30] declaration says: reserve 30 memory locations for storing the characters that comprise the name part of the Label.

This is similar to the DIM statement in BASIC, in that it places a fixed limit on the amount of memory reserved. A word of warning, however: in most cases, a C compiler will not prevent you from writing such a large character string (for example) into this storage that you overrun the space you have reserved. In fact, this is often the cause of visits from the Guru – some program has inadvertently stepped on the wrong part of memory.

The last part of the Label declaration says:

```
LONG zip;
```

I've chosen to use a LONG variable here instead of a string of characters. A LONG variable is a large integer, whose value spans 4 bytes of memory storage. If you specified a SHORT integer here, you would have been limited to a zip code whose value was 32767 or less. A USHORT would have allowed a zip code whose value was as much as 65535. So we have to go all the way to a LONG value to get enough bits to store a zip out to 99999.

So now we have a data structure defined as a "Label". Let's see how it can be used in programs.

Suppose you have a particular group of people who are interested in greeting cards, and another group of people who are interested in business cards. If the Label declaration is already included as part of your C language program, with:

```
#include "label.h"
```

(If the file "label.h" contains the definition of the "Label" structure, this statement makes the definition available to your program.)

then you could tell the compiler that you were working with two different groups of people by declaring the two following data structure arrays:

```
struct Label greetings[100];
struct Label business[100];
```

This tells the C compiler to reserve space for 100 data structures of the size of Label, and to call them "greetings"; and to reserve 100 data structures of the size of Label, and to call them "business". Notice that each Label structure is 120 memory locations long – 14 for the "LNode", 30 for the name, 50 for the address, 20 for the city, 2 for the state, and 4 for the zipcode.

Note here that, as in the BASIC program, the declaration you see for greetings and business are just like the BASIC DIM statements, and would restrict the program to using only that maximum number of data items.

Locating an Element

When building a data structure in memory, the compiler always puts the items in consecutive memory locations. To locate a particular element (member) of a structure, it uses the definitions you provide to calculate the address at which the element resides.

In the example above, if the first element of the "greetings" array starts at location 10000 (decimal), then the first 14 memory locations beginning at that location will be filled with stuff related to the LNode part of the structure. The address of the first element of the name will therefore be located at 10014 decimal (locations 10000 to 10013 inclusive are occupied by the Node part.)

What's Next?

So far you've seen static allocation of data structures (that is, we declared two arrays, greetings and business, each of which is an array of the set of data elements defined as a "Label"). That method still was limited in that the number of elements we can access is defined at the time the program is first compiled.

Since we'd like to leave the door open to add new elements, we want to allocate the elements dynamically instead of statically. Therefore we'll look at how to ask the operating system to give us a block of memory, and how to fix it so that this block is made part of a *list* of blocks that we control.

Dynamic Allocation

The Amiga provides a function called AllocMem that is used to request a block of memory from the operating system. If your call to this function returns a non-zero value, the operating system has indeed given you exclusive use of a block of memory. With the Amiga, it also means that the operating system does not know about that block any more. When your program exits, you have to arrange to return that memory to the operating system so that it can be allocated to another task at a later time. We'll look at memory deallocation in more detail below.

Here is an example that shows a call to AllocMem. The example introduces another new concept (new to this article anyway): the pointer variable. A pointer is a variable that contains the address of something. You have to declare what kind of thing it is pointing to (a structure, for example), so that the compiler knows how to use the pointer as a means of referring to that thing.

```
struct Label *Lpointer;

    /* declares a pointer to a Label structure */

/* . . . later in the program */

Lpointer = (struct Label *)AllocMem(sizeof(struct Label),
        MEMF_CLEAR);
```

This example says: "Give me a block of memory that is the size of the Label structure, and put the address of the lowest memory location in that block into the pointer named Lpointer". The notation:

```
(struct Label *)
```

is called a "type cast". When you are trying to equate one thing to another, the C compiler checks that the two items are of the same or compatible types. If they are not compatible, the compiler complains and may not finish compiling your program.

The type cast is needed because the function AllocMem returns a different kind of pointer. The type cast tells the compiler that it should not worry about the kind of value returned by AllocMem and simply to treat it as though it is a pointer to a Label. Then when the compiler sees the equals sign, it will be equating two items of the same type. That makes the compiler happy.

The parameter named MEMF_CLEAR says that the memory allocator should fill the block of memory with zeroes before it returns to us with the address.

Keeping Track of the Entities

Now that we have a block of memory, how are we to keep track of it? Recall that we'll have to give it back when the program exits or else the operating system will not be able to re-assign it after our program has finished running. How about adding this memory block to a list, then using this list to represent our array of people? Let's create a list named Greetings:

```
struct List Greetings; /* declare a List structure */

/* . . . later in program */

NewList(&Greetings);  /* give address of List to NewList
                       * to allow it to be initialized  */
```

(It says "later in the program" because in C, the declarations of the types of entities comes first, with executable statements following.)

Now, *this* is where the "Lnode" part of our data structure comes in. The system list handling routines each expect that whatever item is handed to them will refer to a block of memory that is structured as a Node. A Node contains data that enable lists to be created, such as a pointer to the previous item in the list, a pointer to the next item in the list, and a priority number that tells the list handler where to position an item within a list.

We need do nothing to this Node part if we simply want to add our memory block onto the list. Because MEMF_CLEAR was used, all items in the Node part of the data structure will be zeroes, and will therefore be treated alike.

If this newly-allocated item is to be a part of the Greetings list, let's add it to that list:

```
AddTail(&Greetings, Lpointer);
```

AddTail is a function that adds items to the end of a list. It accepts two pointers or address values: the first is the address of the list onto which it is adding an element, the second is the address of the element to add. The ampersand (&) in front of the word Greetings tells the compiler to use the address of the data structure (the List) Greetings as the first parameter. Since Lpointer contains the address of the element, this part is okay too.

Now let's see how to install the data into the structure, using a pointer. If Lpointer contains the address of the first element of a Label data structure, then the address of the first element of the name is:

```
&Lpointer->name[0]
```

This is the address (because of the "&") of an element within the data structure called "Label" (because Lpointer is declared to be a pointer to a Label data structure). The forward arrow ("->") means that a specific member of the structure is being referred to, in this case the name array. We want the address of array element 0, the first element. (In C, indexes of arrays begin with element 0 and go to "size_of_array – 1" rather than from 1 to size_of_array.)

Since there are several elements in this data structure, it would be easier to read the program if we could employ convenient names for the individual elements, instead of using the notation "&Lpointer. . ." and so on. So let's take advantage of yet another of a C compiler's standard features, the C preprocessor. This lets us define terms using a "#define" statement; when the preprocessor sees those terms, it replaces them with the definition we have provided. This makes our program easier to read, yet still allows C to know exactly which variables we mean to use.

Our defines, for pointers, can be:

```
#define NAMEG      &Lpointer->name[0]
#define ADDRESSG   &Lpointer->address[0]
#define CITYG      &Lpointer->city[0]
#define STATEG     &Lpointer->state[0]
#define ZIPG       &Lpointer->zip
```

Unlike the BASIC example, we no longer have a specific INDEX variable to use. Instead, we must rely on the value contained in Lpointer having a legitimate value – that of one of the Label data structures we have created. If Lpointer contains a proper value, then the defined items NAMEG, ADDRESSG and so on can be used as though they contained the address of the name element, the address element and so on.

Putting Things Into The Data Structure

In C, copying strings from one location to another is not necessarily possible by direct assignment – that is, by simply using an equals sign (as you could in BASIC). Instead, you have to use a function (supplied in a "library" of functions with the compiler) to do the copying. I use "strcpy", which accepts a destination address and a source address as its parameters. If I have allocated a Greeting structure, and have a good value (nonzero) as my Lpointer, then I could use:

```
strcpy(NAMEG, InputString);
```

to copy my input string into the name portion of my data structure. The C preprocessor makes the substitution for the NAMEG definition at compile time, and establishes the correct address into which the string is to be copied.

At the end of this article, I have provided a greatly abbreviated example in which only a name and a phone number string are accepted by the program into a list of data structures. This example makes no attempt to use any fancy list-handling functions to sequence the items in the list, and no attempt to save the data to a file. It is only a simple, but runnable, example of the dynamic creation and deletion of data structures using a few of the simpler list-handling functions.

Why Should I Be Interested In Lists?

In this article I have provided a tiny introduction to lists, hoping to whet your appetite to find out more. You see, almost everything in Exec, the controlling program for the Amiga's multitasking system, is built with lists.

There are lists of memory areas that are free to be allocated to running programs (that's how the memory allocator works – by grabbing a block of memory from its list of free areas and giving it to your program to use).

There are lists of devices that AmigaDOS knows how to talk to, lists of libraries of routines that your functions can call, lists of tasks that are ready to run, lists of tasks that are waiting for things to happen before they can run again, and lists of message ports to which messages can be sent between running tasks.

Lots of lists for lots of purposes. If you've found this to be interesting, check out some of the books about the Amiga's operating system. They'll tell you more.

```
/* phonelist.c – tiny data structure example */

/* note: start this program from a CLI:

      1> phonelist

   DO NOT say "run phonelist"
*/

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/memory.h"
#include "stdio.h"

extern APTR AllocMem(), RemHead();

struct PhoneEntry {
   struct Node Pnode;
   char  name[41];
   char  phonenum[31];
};

struct PhoneEntry *pe;
struct PhoneEntry *nextpe;

#define NAMEP     &pe->name[0]       /* address of name part     */
#define NUMP      &pe->phonenum[0]   /* address of number part   */

main()
{
   struct List Plist;

   char buffer1[256];          /* place to store input name  */
   char buffer2[256];          /* for phone number           */
   int c, quit;
   char mode;                  /* determine what to do.      */

   NewList(&Plist);            /* initialize list header     */

   printf("List empty, enter data? (y/n) ");
   gets(buffer1);
   c = buffer1[0];
   if (c == 'n') exit(0);      /* end if no data             */

   mode = 'a';                 /* start add-mode             */
   quit = 0;                   /* don't quit so soon         */
   while (quit == 0) {         /* outer loop: get then show data  */
      switch(mode) {
         case 'a':             /* add mode                   */
            while (1) {        /* forever till break out of loop  */
               printf("Enter name (max 40 characters)\n");
               printf("or 'Q' to quit entry mode\n");
               printf("then press RETURN\n");
               gets(buffer1); /* get a string */

               if (strlen(buffer1) == 1 && buffer1[0] == 'Q') break;
               /* do something else if done entering */

               printf("Now enter phone number; then press RETURN\n");
               gets(buffer2);

               pe = (struct PhoneEntry *)
                  AllocMem(sizeof(struct PhoneEntry),
                  MEMF_CLEAR);

               if (pe != NULL) {

                  buffer1[40] = '\0';        /* truncate input string      */
                  strcpy(NAMEP, buffer1);
                  buffer2[30] = '\0';        /* this one too, just in case */
                  strcpy(NUMP, buffer2);
                  AddTail(&Plist, pe);       /* add to end of list         */
               }
               /* does this forever, till 'Q' "break" */
            }
            printf("Here are all entries you made so far:\n\n");

            pe = (struct PhoneEntry *)Plist.lh_Head;
            while (pe->Pnode.ln_Succ != NULL) {
               printf("%s : %s\n", NAMEP, NUMP);
               pe = (struct PhoneEntry *)pe->Pnode.ln_Succ;
               /* point to next one */
            }
            break;

         case 'd':          /* delete mode */
            pe = (struct PhoneEntry *)Plist.lh_Head;
            while (pe->Pnode.ln_Succ != NULL) {
               nextpe=(struct PhoneEntry *)pe->Pnode.ln_Succ;
               /* point to next one */
               printf("Current Entry is:\n");
               printf("%s : %s\n",NAMEP,NUMP);
               printf("Delete it? (y/n)");
               gets(buffer1);
               if (buffer1[0] == 'y') {
                  Remove(pe);
                  FreeMem(pe, sizeof(struct PhoneEntry));
               }
               pe = nextpe;          /* now report on next one  */
            }
            break;

         case 'q':                   /* quit mode               */
            quit = 1;
            continue;

         default:                    /* anythin' else           */
            break;
         }   /* end of switch on mode */
      printf("Add more (a), Delete items (d), quit (q)\n");
      printf("(a/d/q)");
      gets(buffer1);
      mode = buffer1[0];
   }    /* end of outer forever loop */

cleanup:  /* come here when ready to quit */
   while (1) {
      /* pull from list, one at a time */
      pe = (struct PhoneEntry *)RemHead(&Plist, pe);
      if (pe != NULL)
         FreeMem(pe, sizeof(struct PhoneEntry));
      else
         break;
   }
} /* end of main */
```

In the Beginning was CAOS

by Andy Finkel

© Copyright 1988 by Andy Finkel, Commodore-Amiga, Inc.

. . .what would the Amiga have been like with the original DOS?

Andy Finkel is the current Manager of Amiga Software at Commodore. He has been involved with Commodore machines at Commodore since before the introduction of the VIC-20.

As most of you know, AmigaDOS was not the first choice for the top-level OS/DOS on the Amiga computer. What we now call AmigaDOS was really the backup DOS, based on an already existing OS known as Tripos (developed at the University of Cambridge Computer Laboratory by the TRIPOS Research Group, and converted with amazing speed by Metacomco's Dr. Tim King and his band of programmers). When the original, intended, designed-for-the-Amiga DOS failed to materialize (in what would no doubt be an interesting story in itself), the Amiga was launched with AmigaDOS, and the rest is history, so to speak.

Once in a while, someone suggests that the original DOS be written according to the original specs. This was even proposed by some members of the Amiga team, but Amiga management decided that it wouldn't be possible to complete the DOS and still launch the Amiga on time, especially since the software guys had already given up weekends at home. And *going* home. And sleeping.

We're left with a big question. . . what would the Amiga have been like if we could have used the original DOS? What would the character of the Amiga have been? Would life have then been perfect?

There's really no way of knowing, of course. But if I stopped here, this would be a really short article, right? So why not take a quick look at the specs, and maybe do a little comparison with AmigaDOS. . .

Because these questions greatly interest developers for some reason, a small mention of CAOS on USENET brought a flurry of electronic mail, all asking for more detail (all right, they wanted the specifications). So I decided to write this – it seemed like more fun than the article I was working on ("A Guide to DOSBase", or "Why is this library different from all other libraries?") so I'll save that for another time. Ideally, this article should have been written by one of the Amiga people who spec'd CAOS (primarily Carl Sassenrath who also wrote Exec) way back in the dim reaches of time (1984). However, as they've been quiet about it all this time, and I have played with the prototype (do you know what a pain a Sage can be to set up?) it's time. Anyway, the specs are from the CAOS draft document.

CAOS stands for the Commodore Amiga Operating System. Its purpose was (good place for a quote from the draft specification here):

> *It is a small, full-bodied operating system, functioning on a small (but great) machine bound for the consumer marketplace. Its primary purpose is not to provide a sophisticated software development environment – rather it provides a foundation for pleasant and useful application programs. This is not to say that it neglects the programmer's needs. Externally it is meant for end users; internally, however, it is designed to support the complex demands of most applications. It provides extra rich functionality for an operating system of its class.*

The goals of CAOS are:

functionality	
usability	*(easy to use at both outer and inner interfaces)*
programmability	*(interactive batch language, rich set of programmer functions)*
extensibility	*(modular and extensible)*
reliability	
performance	
size	*(reasonably compact)*
maintainability	

CAOS was always meant to be built on top of Exec (in much the same way that AmigaDOS is built on top of Exec). Exec would provide the low level services. It would initialize the hardware, bring the system software up, handle interrupts and exceptions, do the low-level device control (like the trackdisk.device), provide system error-handling, supply the low-level timing routines and semaphores, provide some useful utility functions, and handle the basic multitasking.

This sounds familiar, right? Yes, it's true. AmigaDOS depends on Exec for the same functions.

What would CAOS itself have given? How about multiprocessing, a file system, memory management, and commands? Basically, the same things that AmigaDOS gives us – not surprising, as AmigaDOS had to step into the shoes of CAOS. Both CAOS and AmigaDOS have their own special strengths, though. For instance, CAOS also did memory management to a greater extent than AmigaDOS, and would have had resource tracking.

There are three (well, at least three) general interesting areas to look at about CAOS: Multiprocessing, the File System and Memory Management. And then there's also the projected commands (for fooling around with files and similar operations.)

We might as well start with the easy ones. . . multiprocessing and memory management. You may wonder why multiprocessing is one of the simple ones. It's because multiprocessing (at least in general outline) is much the same as multiprocessing under AmigaDOS.

Processes

CAOS processes are built around Exec tasks. CAOS depends heavily on the Exec data structures and control primitives for multiprocessing (sound familiar?) A CAOS Process structure is built on top of an Exec Task structure. Exec handles the context switching, intertask com-

munication, synchronization, queuing, and mutual exclusion. As with AmigaDOS, only Processes can talk to the file system.

The CAOS Process structure contains (along with the Task structure) information about its stack, the program code, the program data, resource tracking, and exception code. (Once again, except for the 'resource tracking' area, it sounds much like the AmigaDOS process information, right?)

The resource tracking is a key difference. CAOS was intended to keep a linked list containing blocks of resources used by the process – file control blocks, I/O blocks, message ports, libraries, memory usage, shared data, overlays and so on. (This part of CAOS lagged behind the rest; which may explain why we don't have even a semblance of it now.)

Processes run in user mode, using a single (user mode style) stack. Except for context switches, which run in supervisor mode (using the supervisor stack) and of course interrupts, everything was going to stick to user mode. OS system routines are well behaved (they don't mess with the stack), and use well defined amounts of stack space (they don't allocate large areas for local variables or recurse). Now, that's a big difference. AmigaDOS makes extensive use of the stack for its operations. Oh well.

There are three ways to create a process: create it from scratch, allocating new data structures and code; clone it from the parent process, making new data structures, but sharing the parent's code; or chain from the parent, and replace it.

The general description of process creation is:

1) Allocate the process structure
2) Allocate the stack
3) Open standard in, out, and error
4) Load the code, and make it ready to execute (relocate it)
5) Load static data
6) Translate the package file (if one exists), and create a package control block
7) Pre-load the package file
8) Initialize exception vectors
9) Add the task to Exec's task list (with AddTask), which puts it on the queue to run

I mentioned package files in the above list. What exactly are they? Simply this: a package file is a method used to describe the load and run time memory environment for an application by giving the user a way of binding application program and data files in an easy to manage form. This is something we lost when we went to AmigaDOS, and it's something I think we should look at putting in.

Memory Management

Another important task (sorry) that CAOS was intended to perform was memory management. Looking at the current OS one may note that Exec has a rich set of primitives for dealing with memory. CAOS added some additional functionality that AmigaDOS doesn't have (and some that it does).

CAOS basically provided well-managed regions of memory. Within this region of memory, the CAOS memory manager would reign supreme (somewhat like grabbing a portion of memory with the Exec AllocEntry call). CAOS would then manage the memory within that sub-region, allocating sections for code, data, program stack and so on. Obviously, fragmentation could set in. Since scatter-loading was not (at the time of the spec I'm using) part of the DOS design, the CAOS memory manager needed a way to cope with fragmentation. If a sub-region became fragmented, CAOS would attempt to perform a compaction and garbage collection of regions. First, it would delete sub-regions marked as volunteer overlay candidates. If this failed to regain enough memory, sub-regions with no users (libraries and devices with accessor counts of zero, for instance) would be deleted. If this failed, a special entry point would be called for relocatable and unloadable subregions. If this failed, the user would be treated to the Guru (or an out of memory requester)

When a program needed to manage memory within the CAOS-allocated subregions, the program would think of memory in terms of segments. Segments could be marked as relocatable or fixed, permanent or reusable. The CAOS memory manager would attempt to compact relocatable segments and delete reusable segments when memory got tight. Non-relocatable segments would be kept together at either the top or bottom of a region, to avoid catastrophic failure of the relocation/compaction algorithm of the memory manager.

Segments also could undergo a crude form of swapping: they could be read in from or written out to a preallocated disk file.

Believe it or not, the AmigaDOS Seglist gives almost as much functionality. Except for the relocation type of segment and the associated memory manager, you can pretty much play these same games with SegLists.

The CAOS file system

Now we get to the hard stuff – the file system. This is the area that should differ most from AmigaDOS. Providing a file system is *the* most important goal of a disk operating system.

CAOS used a file concept just like AmigaDOS – a file being an abstract data object, with a standardized set of access methods, that appears to the user to be a consistent data stream regardless of its underlying storage form. Many, many computer systems have happened on this concept.

CAOS supported four types of files: ordinary files, directory files, image files, and special files.

An *ordinary* file is used for data storage. The data can be anything – ASCII, binary, whatever. There is no system-imposed structure on the contents of an ordinary file.

Ordinary files are typed as well, to aid CAOS and the user in understanding the files' contents. An ordinary file might be typed as containing graphic information, text, program code or program data, for instance. In the case of an ordinary file of text type, the only limit imposed is that no text line shall be longer than 512 bytes.

A *directory* is a file used to map file names to actual data. They are treated differently from ordinary files in that there is a system-imposed structure for their contents, and they are protected against overwriting. A directory may contain the name of another directory file, and so on – a tree file system is supported.

Image files provide a symbolic method of binding to system internal data objects. CAOS would use these files to give file system-style

access to internal system objects like libraries, devices, interrupt chains, etc. This would have allowed a standard high-level method of dealing with these low-level objects. The file system would have acted as a symbolic address space.

Image files were considered an 'option' – special pseudo-device files might have been used instead. I really like the concept, though. It's one of the clever things I wish we had with the current DOS. (Fortunately, I think I've got a way to retro-fit this onto AmigaDOS.)

Special files are treated just like ordinary files, but don't generally represent an interface to a storage device. They are used to provide standard access to I/O devices.

CAOS allows file names of up to 30 characters. A path can be specified, using the '/' character. Both absolute (from the volume root) and relative (to the current directory) paths are supported. The reserved file name '.' refers to the current directory. The reserved file name '..' refers to the parent of the specified or current directory. A path (including file name) couldn't be longer than 255 characters.

CAOS supported both hard and soft links. A hard link is just a pointer to the internal data structure of a file (actually, a directory under CAOS would be an array of hard links). A soft link references a file by using the file name and doing a text substitution. (For example, if the file system was asked by a user program to Open a file called "library", which was actually a soft link to "/exec/lib/library", the file system would substitute the name "/exec/lib/library" before doing the Open.)

CAOS maintained useful file information, like a description field (up to 255 characters of information for reference purposes), creation time, update time, link count, permissions (read, write, execute, locked), type (directory, printable, non-printable, and so on), user ID, size, and blocks allocated to the file. Image and special files might not maintain all of the information; defaults – you know, lies – are returned when an application program interrogates the file system about one of these files.

Files may be opened for read, write, or append operations. Block and character operation may be supported. You can also seek to any particular byte in an ordinary file. Image and special files disallow this, though.

System directories, like / (the root), /dev, and /Exec are special directories where the system generally keeps its special files, image files, and so on.

Let's go right to the proposed limits on the CAOS file system (always a popular means of comparison). Maximum file size is one megabyte; maximum name size is 30 characters. Maximum path size (including name) is 255 characters (but might have been increased to 512 with enough demand). Maximum number of files would have been one quarter of the number of blocks (about 440 on a floppy); maximum number of open files would have been 16 per process, and the maximum length of a softlink text substitution would have been the same length as a path.

File System Functions and Commands

There are three types of file operations: high level (safe, simple) file access, low-level file operations, which work directly on the file control block (the higher level functions devolve into calls on these low level routines), and support/maintenance operations (i.e. utilities). Naturally, all file functions are called through the CAOS library interface.

Descriptor-based file operations are the easiest means to access the file system. You Open a file using a name (and path), and in return get a descriptor, which you use for all subsequent file operations.

The functions are Open, Close, Read, Write, Seek, EOF (test for EOF), Size (return the size of the file in bytes), Truncate (I miss this one), Update (write out all dirty buffers to the device associated with the file), Name (return a pointer to the file name), Options, Lock (prevent simultaneous access), Unlock (unlock a previously Locked file), LastError, and Control (miscellaneous file access control).

Low-Level File Access

The high-level file access methods described above are based on a lower level file control block access method. Functions using the file control block are: MapFile (go from a descriptor to a control block), OpenFile (given a name, return a file control block), CloseFile, ReadFile, WriteFile, and Update.

By the way, it was considered possible that the low-level access to files would be removed if all operations were possible using high level calls.

Programs/Utilities

Essentially, the programs would be quite similar to the ones we have now (except, of course, Unix™-like syntax would have been used). We'd have Link, SoftLink, Remove, Rename, FileInfo, SearchDir, CurrentDir, ChangeDir, MakeDir, and so on.

Other File-Related Functions

As with AmigaDOS, CAOS supports volumes to logically map disks, partitions, devices, file networks, and structured memory. Volumes can be accessed when they are online. Disk and network volumes go online once their volume ID structure has been read and verified. (For floppy disks, this happens automatically when the disk is inserted.) When access to a volume is ended, the volume becomes either absent, offline, or critical (as in, You Must Replace Volume. . .)

CAOS maintains volume identification information, like name (up to 30 characters), description, time stamp and write protection status. By using the "special" type of file, it is possible to get direct access to a volume. In the case of a floppy, the special file would give the same type of access (but in a file-system standard manner) as if the trackdisk.device were used directly.

Order out of CAOS?

This, then is a quick overview of CAOS. Let's say it was available now. . . What would we have gained? Personally, I think the main thing would have been a much more integrated system. From these specs you can see that AmigaDOS is really as powerful as CAOS was intended to be. The problem is that AmigaDOS is different from the rest of the Operating System. CAOS would have used the same type of data structures as the OS, the same type of stack, the same languages (C and assembler), and would have made understanding the system easier.

A Programmer's Tour of "arp.library": History, Functions and Future

by Scott Ballantyne
© Copyright 1988 by Scott Ballantyne

. . .As you can imagine, replacing AmigaDOS is a monumental undertaking. . .

Scott Ballantyne is a programmer whose current interests are operating systems and computer language design. Scott is well known as the author of the popular Blazin' Forth for the C64. He lives in Manhattan and may be reached via CIS: 70066,603 or BIX as sdb.

A Brief History of ARP

The ARP (AmigaDOS Replacement Programs) project is about ten months old at this point, but the real history of the project actually starts with the release of the Amiga. Like many of you, those of us involved with ARP were thrilled with the Amiga. We spent hours drawing lines and making filled shapes on the Amiga screen, gloating over the speed of the hardware and the excellence of the graphics support functions. We discovered how to make it beep and talk, and discovered all sorts of wonderful things this box could do. And then we discovered AmigaDOS, the highest level of the Amiga's operating system, and uncovered the worm in the proverbial apple. The history of ARP actually begins with this early distress at the condition of the Amiga's DOS.

Many people have been very vocal about the shortcomings and problems with AmigaDOS, but little has been done to fix the major problems. Complaints and disgruntlement continued, until Charlie Heath of Microsmiths Inc. decided to do something about it. He proposed that everybody start writing replacement programs for the AmigaDOS commands, contributed a few himself to get the ball rolling, and ARP was born.

Like most major software projects involving more than one individual, the early days of ARP were marked by heated discussions over the style of the replacement commands, the user interface, and just about anything you can imagine. It looked for a while like ARP was going to drown in an ocean of conflicting opinion, and equally conflicting code.

Around March of 1987, "arp.library" – the subject of this article – first saw the light of day, and provided a focus for the continued development of ARP that was really remarkable. This, combined with a decision to go with as much AmigaDOS compatibility as possible (largely at the urging of Andy Finkel of Commodore) gave ARP an impetus and a direction that has continued up to the present moment.

As you can imagine, replacing AmigaDOS is a monumental undertaking, and is going to take some time to complete. The current release (which was made public at AmiEXPO in October 1987), consists of a large percentage of very compatible command replacements for the AmigaDOS commands, and an Amiga-style shared library. Altogether, these commands will save you about 20k of disk space, and greatly improve the Amiga CLI environment by providing faster, smaller, and much more powerful commands. In addition, the library (called arp.library, logically enough) provides a set of powerful functions which can be called from any language. Later on, I will be describing a large number of these functions, with examples of how to use them in the C language.

What is wrong with the current DOS

AmigaDOS is written in BCPL, which was popular in North America in the Sixties. People started fixing problems with it, and it evolved into another language called simply C, which I am sure you have all heard about. While there is nothing intrinsically wrong with BCPL as a systems programming language, the particular implementation that was used for writing AmigaDOS has a large set of problems in the Amiga environment. Many problems with BCPL on the Amiga are well known, things like the dreaded BPTR, which must always be multiplied by 4 before being used to access a machine location, the upwardly growing stack which does not map at all well onto the 68000 series CPUs, and some of the worst code generation I personally have ever seen from a compiler for any language. While all of these are serious problems from the standpoint of efficiency, they are not too bad. The real problem with AmigaDOS is that it does not observe Amiga established standards for Amiga system software.

The designers of the Amiga established certain language independent conventions that all Amiga programs are expected to adhere to. These conventions are certainly not restrictive, and there is no reason for them to be ignored. As proof of this, the Amiga has successfully been programmed using such different languages as C, Modula 2, Basic, Lisp, Icon, Forth, Fortran and BCPL. While all these languages are able to call on any of the Amiga system library functions, they are not able to call on the private BCPL library functions, because the BCPL library does not adhere to the established conventions. As a result, there is a large amount of code in the Amiga ROMs that only BCPL programs can access.

Even worse than this is the fact that conventions for program startup on the Amiga are complicated by the need to maintain two different sets of register and stack conventions: one for BCPL, and one for the rest of the world. This has caused serious problems in practice, particularly when it is necessary or convenient to have one program load and run another. This is a useful programming technique on multi-tasking systems, but it is still not widely used on the Amiga, even when it could be used to great effect, because no one is certain exactly what type of program they can expect, a BCPL program or a "normal" one. If you get the wrong one, it's a disaster.

AmigaDOS also has some problems from the standpoint of system level support for multitasking. Certain of the AmigaDOS global data structures are not well designed for shared system use. In particular, the AmigaDOS DevInfo list is not at all protected from being stomped on by more than one program at one time.

From the user standpoint, the BCPL AmigaDOS command programs have an entirely different set of problems. They are inconsistent in command syntax, provide insufficient power for a machine of the Amiga's quality, and are too big and too slow.

What ARP does to fix these problems

ARP fixes the above deficiencies in several different ways. The ARP replacement commands are hand crafted in assembly language, which neatly solves the problems of size and speed. We also made every effort to make each command as consistent and as powerful as possible.

By replacing the contents of the C directory, we have also made a major step in removing the necessity for special BCPL register and stack setups. Programs running in an ARP environment are running in a standard Amiga environment and therefore know what to expect. This will greatly increase the reliability of programs written for the Amiga, and should also shorten development time and program size.

The ARP C directory replacement commands rely heavily on "arp.library", which is a standard Amiga system library that provides the functions necessary to write AmigaDOS style programs easily and efficiently. Since this is a standard Amiga library, any language is welcome to call on it at any time, and we currently provide interface libraries and startup code for Manx and Lattice C compilers, as well as some interface code for Modula II. There is no longer any need for a special BCPL library in ROM.

Arp.library also provides a significant grab bag of 'extra' functions, some of which are designed to fix AmigaDOS problems, such as handling the device list. Other unique and important features of arp.library include resource tracking (we feel this is a particularly important first for the Amiga), environment variables and a full blown, Intuitionized File Requester.

Calling arp.library from C

There are actually two ways you can incorporate arp.library into your C programs. We have provided two linkable libraries (Manx users also get 32-bit versions of these), and two header files, arpfunctions.h and arpbase.h. The source code to all of these is available so you can tailor them to suit your personal needs and preferences.

The two C libraries are called a.lib and arp.lib. A.lib contains all the glue routines that let your compiler hook up with arp.library. Arp.lib contains all the glue routines, and in addition contains arp.library specific startup code that opens arp.library and uses ARP's memory allocators and argument parsers to do standard AmigaDOS style command line processing, including the standard interactive '?' as well as ARP's enhanced command line help options.

Using the supplied startup code: "Startup.c"

The example program in Listing 1, "Startup.c", demonstrates how you might use the ARP-supplied startup code to write a program that expects commands from the CLI environment, and presents the familiar AmigaDOS template and help features. Compile the program, link with arp.lib (you will also need to link with your compiler-supplied library), and then play with it a bit to get a feel for how it works.

The startup code calls ARP's command line parser, known as GADS(). GADS() stands for Get Arguments Dos Style, and that is essentially what it does. You establish the appropriate command template by setting the global variable CLI_Template to the appropriate character string, add an optional help line by setting CLI_Help, and let GADS() and the startup code do the hard part of parsing the command line into its component parts.

If the user supplies escapes inside of quotes, GADS() will properly handle these as well. GADS() will also handle certain command line errors, and the startup code will display an error message describing the error and abort the program, so a large amount of error handling is also done for you. If you don't set values for CLI_Template or CLI_Help, you will get the default settings.

All you really have to do is set up a proper template. While only you can decide what is the best form of command template for your programs, some good guidelines are:

1) Don't use whitespace (tabs and spaces) in keywords.
2) Use the '=' idiom to provide short forms for switches or options, for example:

   "FROM/A,E=EVERYTHING/S,Q=QUIET/S"

3) The above template will allow the user to type an E to represent the switch EVERYTHING, and a Q for QUIET.
4) Generally, if you are expecting filenames, try to use as many as you can, at least 11, or even 15 if it makes sense. You can specify filenames without having to give a description for each one by simply using commas alone:

   ",,,"

The above template allows up to four separate commands.

For more information, see the supplied example program, and the manual page for the GADS() argument parser in the ARP Programmer's Docs.

By using GADS() (either through the supplied startup code, or directly), you can easily parse command lines, and also help to present a uniform command line interface to the user of your program, who is already accustomed to the Amiga style command templates.

One other advantage of using the supplied startup code is free access to IntuitionBase and GfxBase. Arp.library opens these libraries as part of its own startup, and these are valid library pointers for the life of your program. The arp.lib C startup code copies these pointers into the expected variables for your use. You might also be interested in comparing program sizes. Using the arp.lib startup code gives you a program size of under 3000 bytes. The same program compiled with the standard C startup code, and using the standard C printf() instead of the arp.library Printf(), results in a program of over 6000 bytes (these sizes are using Manx).

Using ARP without the startup: "Nostartup.c"

You have the option of linking with the standard C runtime code supplied with your compiler, or opening arp.library yourself. A typical example of this is given in "Nostartup.c" (Listing 2). Worthy of note is the method of obtaining the intuition.library and graphics.library pointers from ArpBase. This program should be linked with a.lib and then whatever compiler-dependent libraries you need, in that order.

Program exits using arp.library

Arp.library provides its own program exit routine, called ArpExit(). This function will release all tracked resources and close the library for you.

However, generally speaking, you should not use ArpExit() directly from a C program. The safest and best way to exit from a C program that is using ARP, no matter which version of the .lib files you have linked with, is to use your compiler's exit() function, or to simply fall off the end of the world. If you are not using the supplied ARP startup code, then you should first CloseLibrary(ArpBase). If you are using the supplied arp startup code, then you should *not* CloseLibrary(ArpBase).

The considerations here involve interactions with your compiler's own linkable C libraries. These libraries may have obtained resources on your behalf which the exit() routine is expected to free. If you do an immediate ArpExit(), these resources will not be returned for you. In addition, if you expect to run your program from Workbench, it is important that the exit code Reply() to the Workbench startup message. None of this will happen if you call ArpExit() (or if you call the similar AmigaDOS call Exit()).

If you are using your own custom startup code, and you are not linking with any C libraries, then you may safely use ArpExit().

Because of its smaller size, the ability to use all of the ARP functions as well as the standard C library functions, its use of the ARP command line parser, and since it will run as well from Workbench as it will from the CLI, we will be using the ARP startup code for all the programs in this article, and feel that this is generally the method of choice for C programs on the Amiga.

The meat and potatoes of arp.library: Utility Functions

ARP provides several functions that do useful but not very exciting things in a highly efficient way. If you have programmed in C before, many of these will be familiar to you, although some care is needed, since the behavior may be subtly different from what you expect. In some ways, these functions are more significant for assembly language programmers, since they provide an easy way to do things like string comparisons and display formatting which ordinarily require custom subroutines. Because these functions are already quite familiar, I will touch on them only briefly here. These are generally given names similar to their C library counterparts, but the first letter of the arp.library name is capitalized to prevent c.lib collisions.

Arp.library contains very standard C-style Printf(), FPrintf() and Puts() functions. If you need to display formatted output, you should definitely consider using the arp.library display functions, since the typical C compiler equivalents usually increase program size by several thousand bytes. The only things the arp.library functions will not do is display floating point numbers correctly. If you need to display floating point values, then you are stuck with your compiler's version of these routines. Another advantage to using arp.library's version of these routines is that they are "safe". If you should happen to call an arp.library Printf() or Puts() from a program running under Workbench, you will be pleased to discover that these functions will not crash. The same is not true of the C compiler equivalents, as many people have discovered to their sorrow.

One thing to watch out for when passing values to Printf() to format is the fact that C will convert integer arguments to its default integer type. This can cause certain problems if you are not aware of it. For example, the following Printf() statement will function correctly only using the Manx C compiler set for 16-bit default integers:

```
Printf("This should display 10 = %d", 10);
```

Using Lattice or the +L switch under Manx, you should use the following form of the above expression:

```
Printf("This should display 10 = %ld", 10);
```

This is because the compilers will promote the integer argument to a long, but the arp.library routine will not be aware of this – it assumes you know what arguments you are passing to it.

Other C style utility functions in arp.library include string comparison routines (Strcmp and Strncmp), case conversion (Toupper), string to integer conversion (Atol), and a lightning fast, non-recursive quick sort function (QSort). Note that the string comparison routines were designed to work with AmigaDOS, and are therefore not sensitive to case. All of these functions are used every day by C programmers, and these implementations are very fast, very efficient, and since they are in a shared library, they keep your program size down considerably.

Arp.library is not without glamour: FileRequest()

The arp.library file requester is a vastly enhanced version of the Heath file requester that was used in many of the early and important products for the Amiga. You may have seen it in Microsmiths' TxEd, or in some of the Aegis products such as Draw or Images. It has always been regarded as one of the best file requesters for the Amiga, and its inclusion in arp.library makes it available for the price of a function call.

It is extremely easy to use the file requester. You need to initialize a very simple structure called, reasonably enough, a FileRequester structure, and submit this to the FileRequest() function. If the user clicks the cancel gadget, the return from the FileRequest() function is NULL, otherwise, it is a pointer to the filename field in the FileRequester structure. Here is the file requester struct, as defined in arpbase.h:

```
    BYTE  *fr_Hail;          /* Hailing text */
    BYTE  *fr_File;          /* Pointer to storage for filename */
    BYTE  *fr_Dir;           /* Pointer to storage for Directory */
    struct Window *fr_Window;        /* Pointer to window or NULL */
    SHORT fr_Flags;          /* Not used in this release */
    VOID  (*fr_WildFunc)(); /* Not used in this release */
    VOID  (*fr_MsgFunc)(); /* Not used in this release */
};
```

The variables marked as "Not used in this release" are for future expansion, and should be initialized to NULL. The fr_Hail variable points to text that will be displayed in the body of the requester. This can be the name of your program, or a brief instruction, whatever you like.

The fr_File and fr_Dir variables should point to character arrays for the FileRequest() function to store the user's selections for Directory and File. If you get a non-null return from FileRequest(), you should look here for the choices. The fr_File array should be large enough to hold one filename, or FCHARS+1 length long. fr_Dir should be large enough to hold the longest possible pathname you expect. It wise to be generous in allotting space for the fr_Dir array.

You can initialize the fr_File and fr_Dir arrays to default values. These will be the directory searched and the file displayed in the filename select box when the FileRequester starts up. A short example of how to use FileRequest() is in "FileRequester.c" (Listing 3).

You should note that FileRequest() will search directories using wildcards. For example, if the user were to specify a drawer name of "DF0:C/E*" the file requester would only display files in DF0:C which start with the letter 'E'. In this case, should a user select a file, the directory string returned to you will include the wildcard characters.

Note also that it is possible to get a NULL filename returned to you. This can happen if the user clicks on the OK gadget without selecting a filename first. Your program should be set up to handle these specific cases reasonably. You should be able to get a clear idea of how to use FileRequest() by playing around with the "FileRequester.c" demo, and looking at what it is returning to you.

Let arp.library track your resources

One of the great unsolved problems of Amiga programming is the freeing of allocated resources. Arp.library contains functions that will allow you to cleanly track the most commonly allocated resources with a simple function call, and also provides direct access to the underlying tracking mechanism so that you can track pretty much what you want.

The basic idea behind the resource tracking functions is quite simple. When you first call a tracking function, your task is allocated its own resource list (ResList in ARPspeak), and the resource you are allocated is chained into this list. Further calls to tracking functions continue to chain resource nodes into your tasks ResList. Eventually, your program will exit, and CloseLibrary(ArpBase). When this occurs (or when ArpExit() is called), arp.library will notice that you have allocated resources hanging around, and will free these things for you before shutting down. Thus, with the arp.lib startup code, all you need to do to exit and free all allocated resources is to call exit().

For detailed information on ARP resource tracking, see the sidebar.

ARP Goes Wild: Searching the filesystem for patterns

If you have been using the ARP replacement commands, you are already aware that one of the most powerful and pleasant features of the new command set is the ability to use wildcards or patterns in commands like Type, Rename, and Protect; in fact, most ARP commands support wildcards. Not only does ARP support the full set of AmigaDOS wildcards, we also support the more standard wildcards many of us grew up with. You are not forced to make a choice, you can use the set you are comfortable with. Using the arp.library directory search functions, you can easily add this same kind of pattern matching to your own programs.

For specifics about using pattern matching in ARP, see the sidebar.

Using DA Lists to find out about devices.

There are many times when it is necessary to find out various details about the specific hardware setup your program is running on. For example, your program may be able to take advantage of multiple disk drives, or require specific logical assignments to be made, or even need to check for whether or not a specific volume is currently in a disk drive or not.

All of this information is maintained currently in a long and complicated list called the DevInfo list. It is not a simple matter to decipher the entries in the DevInfo list, but the real problem with accessing it is that there is no multitasking protection for the list: one program could be altering the list at the same time as you are trying to figure out what is on it.

Arp.library provides the AddDADevs() function which will create a DA list of whatever class of devices you are specifically interested in. For example, you might ask AddDADevs() what disk devices are available, and it will create a list of all devices that support a filing system. Or you could ask it for a list of all known volumes – it will then build a list of all volumes, and indicate (in the de_Type field) whether or not they are currently inserted in a disk drive.

For an example of how to use AddDADevs(), look at the example "Devices.c" (Listing 4). This program simply displays the devices found. Using AddDADevs() to find out what environment you are hanging out in is far superior to hard–coding device names or assuming that everybody is going to have only one disk drive.

Low Level DOS-Like Functions.

Arp.library contains functions that make accessing ordinarily difficult AmigaDOS features much simpler, and in some cases contains routines that appear to be missing from the current implementation of AmigaDOS. Examples of the first type of function are routines that aid in the initialization and sending of DOS packets, and functions that will generate a complete pathname from a file lock. Examples of the second type of function are a function call to perform a logical assignment, and a function to run another program, setting its output and input file handles.

Sending Packets

On a very low level of its operation, AmigaDOS uses packets to transfer information between its various processes. For example, when you use the AmigaDOS function Write() to send data to a file, at some point the Write() function call gets translated into a packet which is sent to the file system process that knows about your file. In turn, the file system will return the packet to the sender (your process) and your program will continue. There is usually no reason to descend to this level of DOS programming, but there are a few things that can only be done at this level. Putting the console into RAW character mode and performing asynchronous file I/O are two examples. *(For more on packets and asynchronous file I/O, see Matt Dillon's article in this issue, "The Packet Interface to AmigaDOS, in C" – CZ.)*

Arp.library contains two functions for handling packets. The first is SendPacket(), which sends a packet to a specific handler, and then waits for its return. It returns the Result1 field of the packet to you, and the Result2 field is available by calling IoErr(), or in register D1 if you are writing in assembly language. Here is an example of using SendPacket() to put the console into raw mode:

```
ULONG args[7]; /* Packet arguments */
args[0] = -1;    /* set raw */
res1    = SendPacket( ACTION_SCREEN_MODE, args,
                FindTask(0L)->pr_ConsoleTask);
```

The second function is provided to initialize standard packets, and is very useful in implementing asynchronous packet communications. Other functions that relate to the sending and receiving of asynchronous packets are CreatePort() and DeletePort(), which are essentially the familiar C library functions of the same name.

Logical Assignments and Environment Variables

Logical assignments are a useful element of the existing Amiga environment, and it is therefore unfortunate and somewhat strange that AmigaDOS does not supply a function call to do this from inside a program. ARP leaps to the rescue with its Assign() function call. It is simplicity itself to use. For example, to assign the disk in drive one the name DODAH:

```
Return = Assign("DODAH", "DF1:");
```

. . .then check the return code (return values for Assign() are defined in arpbase.h as usual). To cancel an assignment:

```
Return = Assign("DODAH", 0L);
```

Note that attempts to cancel assignments can fail also; you should check the return value from these calls as well. A common mistake I enjoy making with the Assign() function call is to try to cancel an assignment this way:

```
Return = Assign("DODAH","");
```

Unfortunately, this results in DODAH referring to the current directory.

Assign() is quite safe and easy to use. You can repeatedly make the same assignment without ill effect, and you can also change the assignment of a logical name without cancelling it first.

As convenient as logical assignments are for dealing with directories and files, they cannot do everything. A common mechanism in modern operating systems is the Environment Variable. Unfortunately, environment variables were not part of the original AmigaDOS package. It was perhaps thought that logical assignments could do it all, but in practice assignments have not proven flexible enough.

Consider the case of an assembler or compiler that needs to look in several directories for its header files. It is possible to do something like:

```
Assign INCLUDE: SYS:INCLUDE
```

. . .and have the assembler written so that it always uses the INCLUDE logical name in searching for header files, but what if you want to search more than one directory? The assign mechanism breaks down at this point, but this could be easily done with environment variables. In fact, the Manx C compiler/assembler package does just this. (Note to Manx users: Since Manx was the first to implement environment variables for the Amiga, the ARP implementation has tried to be compatible with them. You can use the ARP function calls and the ARP Set command and still continue to use your Manx software.)

As another example of the usefulness of environment variables, consider the problem of sending non-filesystem data to a program or group of programs. For example, a copy program might look for an environment variable called OVERWRITE. If OVERWRITE were true, it would copy over existing files if asked to; if OVERWRITE were false, it would not. You can also use environment variables to send numeric data to programs, to establish default values of one sort or another, etc. Many kinds of information may be passed using environment variables that logical assignments, convenient as they are, were not designed to handle.

A user sets the value of an environment variable by using the Set command, whose syntax is:

```
Set VAR=VAL VAR2=VAL2
```

This will give VAR the value of VAL, and VAR2 the value of VAL2. Environment variables may be removed from the environment by not specifying a value:

```
Set VAR= VAR2=
```

You access these variables using the Getenv() and Setenv() function calls. All values of environment variables are stored as ASCII strings, and you are responsible for any parsing or conversion (for example, to numeric data) that needs to be performed. As an example of their use, here is Setenv():

```
Setenv("AMIGA", "GOES BETTER WITH ARP");
```

This call sets an environment variable AMIGA to the value "GOES BETTER WITH ARP". To get the value of an environment variable, you use Getenv():

```
Getenv("AMIGA", buffer, 255L);
```

. . .which fills "buffer" (a character array) with the string assigned to AMIGA. Getenv() also returns a pointer to the value of the variable. This pointer is to arp.library's private storage for this variable and you should not access this area without first using Forbid(). Ordinarily, this will not be necessary, and you can use the return from Getenv() to tell if a value is defined (the return will be NULL if there is no value found).

Running other programs from within your program.

Arp.library provides the function SyncRun(), which will load and run a program and return its exit code to you, or an error indicating that the specified program could not be found or could not be loaded. By convention, programs exiting return a zero if there is no problem, or a value greater than zero for an abnormal termination. SyncRun() will return this value to you. If you get a negative error return from SyncRun(), this indicates that there was a problem in loading the program, for example, the file could not be found, or there was not enough memory.

You can set the default input and output handles for the new program, or you can specify a value of NULL for each, which will cause the newly executing program to inherit your program's default input and output. Here is an example of its use:

```
rc = SyncRun("Echo", "Testing 123", 0L, 0L);

if ( rc > 0)
    Printf("Echo terminated abnormally, rc = %ld", rc);
else if (rc < 0)
    Printf("Cannot load or find Echo");
```

SyncRun() is by far the safest method of doing this sort of thing. First of all, it saves your task's private data, so that you can safely use the private pointers in your task control block without worrying that they might be overwritten by the program you are going to run. In addition, SyncRun() works correctly with arp.library's resource tracking functions, to protect your process's ResList from possible interference with the task your are running.

BCPL programs have bugs which prevent them from receiving the arguments passed to them by SyncRun(). BCPL programs will not crash SyncRun(), but they will execute as though they were not passed a command line. Once BCPL programs join the rest of us in program startup conventions, they too will be able to receive command line arguments from SyncRun().

ARP and BUGS:

No programmer's tour would be complete without a bug report or two. The following problems have turned up in the first release of ARP:

FreeAccess(), GetAccess(), Getenv(), TRAK_GENERIC

These functions are broken in release 31.0; they are fixed in release 31.1, which will be released shortly.

CloseWindowSafely() and TRAK_WINDOW

If you OpenWindow() a window that has the IDCMPFlags variable set to NULL, CloseWindowSafely() will GURU. Only call this function if you have set the IDCMP on a window (by far the most common case).

There are a few additional minor bugs, these will be fixed in release 31.1 also. Developers using ARP should try to release their software with revision 31.1, if possible.

These are the only known bugs in the first release of arp.library.

The End of the Tour: The Future of ARP.

That concludes our tour of arp.library. I couldn't cover every single function in arp, but I did try to cover what I felt would be most immediately useful to programmers in terms of keeping down program size and ability to do things that weren't possible or easily done before.

We will continue for a while yet to work on ARP. The next release, scheduled to come sometime around Christmas, will see bug fixes of current arp.library and command programs, and should also see a full command set, with perhaps a few enhancements. A new CLI-style shell is being worked on, as well as a much enhanced batch language. Charlie Heath of Microsmiths remains the central coordinator of ARP. He is maintaining a mailing list of ARP users and dealers which he will be making available to developers who are using ARP in their products. Since people who are using ARP will tend to be those most interested in quality software on the Amiga, this should provide an important service to everyone.

The ultimate future of ARP of course rests with you, as users and developers, and with Commodore. The initial reception given to ARP by users has been very positive. We are hoping that Commodore will start providing ARP on Workbench disks, and eventually in ROM. While it is too early to say anything definite about this, we did work extremely hard to satisfy the requirements of Andy Finkel at Commodore, who was also very helpful to us throughout the development of ARP. By removing the Amiga's dependence on the BCPL interface, ARP has removed the largest obstacle to future Operating System development. By providing user callable functions such as the GADS() argument parser and the FileRequest() file requester, we have also made significant contributions to standardizing both the CLI and Workbench user interface on the Amiga. Incidentally, ARP is being made available to Commodore free of charge, which should also remove a common obstacle to adding new software: cost. Hopefully, the powers that be will recognize a good thing when they see it, and take advantage of it.

I think I can speak for the other ARP developers as well as the new users of ARP when I say that whatever happens, all the hard work was well worth it. It is really great to be able to fit more stuff on my development and Workbench disks, which I can do thanks to the smaller size of the ARP programs, and being able to use wildcards with programs like Type and Protect is a joy. As a programmer of the Amiga, arp.library provides me with greater functionality and smaller programs. As far as I am concerned, the Amiga goes better with ARP.

• • • • •

How to do Resource Tracking with ARP

The arp.library functions which obtain and track specific resources are these:

ArpOpen()	Obtain and track Files
ArpLock()	Obtain and track Locks
ArpDupLock()	Dup a lock and track it
ArpAlloc()	Obtain PUBLIC memory, initialize to 0 and track.
ArpAllocMem()	Obtain memory à la Exec, and track.

The tracking functions do not use the resource pointers directly, even though that is the value returned by the above functions. Instead, they use an object called a Tracker. A Tracker is the user portion of the tracking node allocated for your resource. It is possible to design your own Tracker if you need extra stuff, but for most applications this shouldn't be necessary, and won't be covered here.

You will need the tracker if you want to free the obtained resource directly yourself later on. To get the tracker for any of the above objects, simply examine the global variable LastTracker immediately after one of the above function calls. You can copy this value someplace safe and then use it later, if you wish, to free the obtained resource by calling the lower level call FreeTrackedItem() directly. Here is a sample sequence:

```
struct DefaultTracker *ftrack;

if ( File = ArpOpen("FileName", MODE_OLDFILE) )
    ftrack = LastTracker;
else
{
    Puts("File didn't open!");
    exit(20);
}

/* Rest of program here, and then, free the item */
FreeTrackedItem(ftrack);  /* Close the file */
```

Of course, in the simplest case, you simply exit the program, and let arp.library close the file for you.

The arp.library tracking mechanism knows how to free many more things than are indicated by the above functions. We provided those functions specifically because these resources are so commonly obtained, it seemed most convenient to have a function that would obtain both the resource and the tracker at the same time.

To track resources other than those shown above, you must obtain a tracker and a resource, and then install the resource into the tracker. Here is a code fragment that illustrates how to use the tracking functions to track an Intuition window:

```
struct Window *Window;
struct DefaultTracker *mytracker;

if ( mytracker = GetTracker( TRAK_WINDOW ) )
{
    if ( Window = OpenWindow( &nw ) )
    {
        mytracker->dt_Object.dt_Resource = (CPTR)Window;
        mytracker->dt_Extra.dt_Window2  = NULL;
    }
    else
    {
        CLEAR_ID( mytracker );
        Puts( "Can't get window!");
```

```
            exit(20);
        }
    }
    else
    {
        Puts( "Can't get tracker!");
        exit(20);
    }
```

The code fragment above illustrates the general approach to take for all of the GetTracker() style function calls.

Obtain a Tracker by calling GetTracker(ID), where ID is one of the TRAK_ID's from arpbase.h.

If the tracker allocation succeeds, then attempt to obtain your resource. If this succeeds, then initialize the DefaultTracker in the appropriate way for this resource. Usually, this will involve stuffing the resource pointer into the dt_Object variable. For certain resources, you may need to do something to the dt_Extra variable. Windows are an example of a resource that requires a value in the dt_Extra field. Each resource that needs the dt_Extra variable will use it differently of course. In the case of windows, the dt_Extra field is used to indicate if more than one window is sharing the same IDCMP. If it is, then the IDCMP will not be closed for this window.

If the resource allocation fails, use the CLEAR_ID() macro from arpbase.h to clear the ID field for the tracker, and then simply exit the program. The tracker will be freed automatically (this is the reason it is usually more efficient to obtain the tracker before you obtain the resource).

You can also specify a function to be called at resource free time by using the TRAK_GENERIC id. This allows you to track additional system resources, custom software structures of your own design, whatever you like.

The resource functions also have the capability to selectively free resources if the system becomes heavily loaded. You specify resources that you can dispense with if memory gets tight by calling FreeAccess() on that item. You can tell if the item is still there, and lock it into memory by calling GetAccess(). Here is a typical sequence, somewhat pseudo coded, since an actual example would be quite large:

```
struct DefaultTracker *dt;
BYTE  *Buffer;

if ( Buffer = (BYTE *)ArpAlloc( HUGE_VALUE ) )
    dt = LastTracker;
else
    exit(20);

/* Here we do stuff with the buffer, and then, when finished
 * we tell the system we would like to keep it, but only if
 * it is not in the way of other resources
 */

FreeAccess( dt );

/* At a later point, when we wish to reuse the resource
 * we must first find out if it is still there, and also
 * lock it into memory while we will be using it – we don't
 * want it to go away while we are in the middle of using it!
 */

Buffer = (BYTE *)GetAccess( dt );

if (Buffer != NULL)
{
    /* do more stuff */
    /* And then */
    FreeAccess( dt );
}
else
{
    /* Item wasn't there – we can try to reallocate it
     * we can free the tracker node, or we can scram
     */
}
```

FreeAccess() and GetAccess() calls may be nested. This allows you to share freeable resources between tasks, processes or modules. Each one can FreeAccess() a resource, or GetAccess() a resource. The system will not free a resource unless none of the tasks are using the resource.

Finally, it is also possible to call the lowest level tracking functions, CreateTaskResList() and FreeTaskResList() to get a new resource list for your task. This is usually done for you when you call one of the tracking functions, but it is also callable directly. One interesting possibility that this opens up is the ability to have nested resource lists. You can write a function that calls CreateTaskResList() at entry, and then calls FreeTaskResList() at exit, to free only those resources allocated by that function (and by all the functions that it called).

ARP Pattern Matching and Wildcards

The directory search functions use a data structure called an Anchor. It is not necessary for you to obtain these Anchor structures, but you need to be aware that they are being obtained on your behalf by the functions that search directories, and you must explicitly free them at the end of a directory search. You can free them all at once by calling the function FreeAnchorChain(), or you may decide to use the arp.library tracking functions as described in the previous section to free them automatically on program exit.

To obtain your initial Anchor, you must allocate and initialize a control structure called an AnchorPath. The AnchorPath structure contains the BCPL structure FileInfoBlock as one of its elements, and so you must insure that the entire AnchorPath is longword aligned. The easiest way to do this is to use the ArpAlloc() function call. Here is the AnchorPath struct, as given in arpbase.h:

```
struct AnchorPath {
    struct Anchor  *ap_Base;
    struct Anchor  *ap_Last;
    LONG    ap_BreakBits;     /* Bits to break on    */
    LONG    ap_FoundBreak;    /* Bits we broke on    */
    ULONG   ap_Length;        /* Size of ap_Buf      */
    struct FileInfoBlock ap_Info;
    BYTE    ap_Buf;
};
```

You can get the filename from the ap_Info structure (see the dos.h file) or you can have the pattern matching functions construct the full path name of the file in ap_Buf. To have the arp.library routines build the full path name, you must extend the AnchorPath structure somewhat to accommodate a larger buffer, and place the buffer size in ap_Length. If you only need the filename as obtained from the ap_Info structure, then you do not need to extend the structure, and you should set

ap_Length to zero. A simple and convenient way to enlarge ap_Buf is to declare your own AnchorPath structure as follows:

```
struct UserAnchor {
    struct    AnchorPath ua_AP;
    BYTE      moremem[255];   /* Extend ap_Buf[]    */
};
```

Once you allocate a UserAnchor structure with ArpAlloc(), you will have an ap_Buf of 256 bytes. If the pattern matching functions find a name that is longer than ap_Length bytes, they return ERROR_BUFFER_OVERFLOW.

The pattern matching functions of ARP are very fast, but whole directory searches can take some time. You can let the user break out of a directory search by initializing the ap_BreakBits variable to the signal bits you wish to respond to. These signal bits are defined in the dos.h file, and correspond to the user hitting the control C D E or F keys, or using the Break program to do the equivalent for a background process. Once the pattern matching routines have detected the user's signal, they will return an ERROR_BREAK to you immediately, with the actual signal value in the ap_FoundBreak variable. This allows you to respond to different control sequences in different ways, as well as select which ones you wish to respond to. For the most common case, CONTROL C checks, you would do:

```
ap_BreakBits |= SIGBREAKF_CTRL_C;
```

. . .to activate control C checking during directory searches.

Once you have obtained and initialized your AnchorPath struct, you simply proceed as follows:

1) Call FindFirst("Pat", AnchorPath) to get the first occurrence of a File or directory that matches "Pat". If FindFirst() returns zero, then there may be more instances of this pattern in the directory.
2) To continue to search the directory for more instances of "Pat", call FindNext(AnchorPath) until it returns nonzero.
3) Call FreeAnchorChain(AnchorPath) and then handle the error return (which might not be an error).

Both FindFirst() and FindNext() return non-zero on error. When this happens you must free the anchor chain, and then handle the error code. This return might indicate that there are no more entries that match this pattern (ERROR_NO_MORE_ENTRIES), it might indicate a user break (ERROR_BREAK) or it could signal an actual IO or buffer overflow error. For an actual programming example, see "WildDemo1.c" in Listing 5.

Sorted Lists

As you probably noticed if you run the WildDemo1 example, the directory search functions return files in the order in which they find them, which might not be the order you want or need them in. To create ordered lists, arp.library provides the AddDANode() function, which creates single threaded lists of DirectoryEntry nodes. A DirectoryEntry looks like this:

```
struct DirectoryEntry {
    struct DirectoryEntry  *de_Next;
    BYTE      de_Type;
    BYTE      de_Flags;
    BYTE      de_Name[1];
};
```

The de_Flags variable is reserved for future expansion, so you should not use it. Note that lists of DirectoryEntry nodes are single threaded like AmigaDOS lists, and not doubly linked, like Exec lists. Doubly linked lists are best when you need random access to the nodes in a list. For ordered lists, where you just start at the list head and go to the end, doubly linked lists add overhead that is not required.

When you use AddDANode() to add a node to a list, it will be added in sorted order based on the de_Name array (AddDANode() allocates extra memory for this array as needed). Additionally, it will order the lists using the de_Type field as a secondary sort key.

Both of these methods are used in our second try at the WildDemo program (see "WildDemo2.c" in Listing 6). This time, instead of displaying the names of the files as we find them, we add them to a list using AddDANode(). We actually want two alphabetized lists, one of directory names and one of filenames, which we get specifying different secondary sort keys. Since the DIRECTORY key define is numerically less than the FILE key define, the directory entries will appear first in the list, in alphabetical order, followed by the alphabetized lists of files.

After the list is made, we display what we have found by moving through the list one node at a time. We once again use our secondary sort keys, but this time to determine what it is we are displaying, files or directories.

Finally, we release the memory and all the nodes of the list by calling FreeDAList().

Selectively freeing Nodes in DA lists

Since FreeDAList() takes a node pointer (instead of a pointer to DA listhead) as its argument, it is possible to free subranges of a DA list. For example, instead of freeing the entire list, we could have released all the nodes associated with filenames by doing:

```
FreeDAList( First_Filename_Node );
```

There are occasions when you might need to selectively free single nodes of a DA list. To do this, you must use the arp.library function DosFreeMem(), since DA list nodes are allocated using DosAllocMem() (that's where the DA in DA list comes from). An important characteristic of DA list nodes is that they are of variable size. DosAllocMem() and DosFreeMem() work together to keep track of variable sized memory blocks using a simple tagged memory scheme already in use by AmigaDOS. By using this technique of variable sized nodes arp.library is able to allocate all the memory for a single node using one call. This is not only efficient, it also helps prevent memory fragmentation, which can be a real problem in lists of this type.

As an example of how to free DA list nodes, this is how the file display loop in "WildDemo2.c" would have been written, if it also were to free the node after it was finished with it:

```
for ( tmp = de->de_Next ; de; tmp = de->de_Next )
{
    columnize( de->de_Name ); /* Use it  */
    DosFreeMem( de );       /* Free it          */
    de = tmp;               /* Get cached node  */
}
```

Note that while AddDANode() does use DirectoryEntry nodes, FreeDaList() does not. FreeDAList() can be used to free any list whose nodes

have been allocated by DosAllocMem() and that have a pointer to the next node in the list as their first variable.

Listing 1: 'Startup.c', an example of argument line parsing using the ARP startup code. You must link with Arp.lib.

```
/* Startup.c -- Example program for using the ARP supplied startup
 *    code for argument parsing.
 *    -=+SDB+=-
 *
 * Copyright (c) 1987, Scott Ballantyne,
 * Use and abuse as you please.
 *
 * Compile normally (whatever that means) and then link with
 * arp.lib and c.lib.
 *
 * Note that when using the supplied C startup code, you do not need to
 * open or close arplibrary.
 *
 * More significant, you also get IntuitionBase and GfxBase initialized for
 * you - no need to open or close these libraries either.
 */

#include <exec/types.h>
#include <libraries/arpbase.h> /* Get our standard header */
#include <arpfunctions.h>      /* Pre declared functions  */

/* Set CLI_Template to the command line template you wish parsed
 * If you don't, then you will get a default command line template.
 * Note: You will get a linker warning when you do this, to the
 * effect that '_CLI_Template is multiply defined'. That just
 * means you are overriding the startup codes defaults. Not to
 * worry.
 */

char *CLI_Template = "FROM/A,TO,OPT/K,COSMIC/S";

/* The arp.library argument parser (GADS) will stuff these in
 * the appropriate spots in the argv array for you. The startup
 * code cleverly figures out how big to make that array.
 *
 * Here is one way to access the args. The most efficient depends
 * on the template, of course.
 */

#define ARG_FROM    1
#define ARG_TO      2
#define ARG_OPT     3
#define ARG_COSMIC  4

/* Note that GADS() style argument parsing will always stuff the approriate
 * argument in the correct spot in the array. As examples, you might try
 * the following inputs to Startup:
 *
 * 1> Startup FROM file1 TO file2 OPT abc COSMIC
 * 1> Startup From File1 file2 COSMIC
 * 1> Startup Cosmic opt abc TO FILE1 FROM FILE2
 *
 * And, as an example of what happens on a command line error, you might
 * try the following:
 *
 * 1> Startup TO File2
 * 1> Startup From File1 File2 Opt
 * 1> Startup From File1 File2 Opt abc Cosmic karmic
 */

/* Set CLI_Help to the extra help string you wish to provide, if you
 * don't provide any, then you get the default. This will be displayed
 * if a user types a question mark after getting the command line template.
 *
 * 1> Startup ?
 * FROM/A,TO,OPT/K,COSMIC/S: ? <- will display the CLI_Help string.
 */

char *CLI_Help = "This is just a demo, relax already.";

/*
 * The startup code will detect an error return from GADS(), and display
 * the appropriate error message (try a few mistakes and see what you get).
 *
 * The instances of /A arguments are handled this way:
 *
 *  If NO arguments were supplied by the user, then it is up to
 *  you to detect the missing arguments.
 *
 *  If a user supplied arguments, but omitted a required one (/A), then
 *  GADS() will detect this, and the startup code will abort.
 *
 * Finally, the capital P Printf() and FPrintf() are arp.libraries formatting
 * functions, they work just like normal 'C'. They keep programs much smaller.
 */

main(argc, argv)
int argc;
char **argv;
{
    if (argc < 2)  /* Check for absence of required argument */
    {
        Puts("Usage: FROM <this> TO <that> OPT <abc> COSMIC<ally>");
        exit(20);
    }
    /* We know we have the next one, since is a /A argument */
    Printf("FROM argument is \"%s\"\n", argv[ARG_FROM]);
    /* Check for others */
    if ( argv[ARG_TO] )
        Printf("TO argument is \"%s\"\n", argv[ARG_TO]);

    if ( argv[ARG_OPT] )
        Printf("OPTions specified are \"%s\"\n", argv[ARG_OPT]);

    /* Arguments specified as switches are not meaningful as strings,
     * if the switch was specified by the user, it's position in the
     * pointer array is occupied by -1L, otherwise, it is 0L.
     */

    if ( argv[ARG_COSMIC] )
        Puts("Program to progress cosmically");
    else
        Puts("Cosmic switch not in effect");

    /* Continue with rest of program here. */
}
```

Listing 2: 'Nostartup.c', an example of using arp.library without the supplied startup code in arp.lib. Link with a.lib, then the usual libraries. Notice that arp.library must be opened explicitly, but the Intuition and Graphics libraries are opened automatically and their library base pointers are set from members of ArpBase.

```
/* nostartup.c -- Example of how to use ARP without using the supplied
 *    startup code.
 *
 * Note that way we pick up IntuitionBase and GfxBase for free.
 *
 * Copyright (c) 1987, Scott Ballantyne
 * use and abuse as you please.
 *
 * Compile and then link with a.lib
 *
 * -+=SDB+=-
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <arpfunctions.h>

VOID *OpenLibrary();
struct ArpBase *ArpBase;   /* Declare our library base variable */

/* We can get intuition and graphics almost for free */
/* You don't need to do this, of course. */

struct Library *IntuitionBase, *GfxBase;

/* Example of how to open ARP */

main()
{
    char buffer[MaxInputBuf];

    if (!(ArpBase = OpenLibrary(ArpName, ArpVersion)) )
        exit(20);
    /* Opened arp!, now get intuitionbase and graphics */
    GfxBase = ArpBase->GfxBase;
    IntuitionBase = ArpBase->IntuiBase;

    Printf("Type something: ");
    ReadLine(buffer);
    Printf("You typed \"%s\"\n", buffer);

    /* This is the recommended sequence for exiting when using arp
     * from 'C'. It is better not to use ArpExit().
     */

    CloseLibrary(ArpBase);
    exit(0);   /* Or fall off end of world */
}
```

Listing 3, 'FileRequest.c': Using ARP's file requester.

```
/* FileRequest.c -- Example program for using the ARP  FileRequest()
 *    function call.
 *    -=+SDB+=-
 *
 * Copyright (c) 1987, Scott Ballantyne
 * Use and abuse as you please.
 *
 * Link with arp.lib
 *
 */
```

```
#include <exec/types.h>
#include <libraries/arpbase.h> /* Standard ARP header file */
#include <arpfunctions.h>      /* Predeclared functions    */

/* Initialize a structure for passing to the arplib's FileRequest()
 * function.
 *
 * You need to supply a file and directory buffer to FileRequest().
 *
 * You can provide both default directories and filenames by initializing
 * the arrays to a string before you submit the structure to FileRequest().
 *
 * For this example, we only initialize the directory string.
 */

#define MAXPATH    ((FCHARS * 10) + DSIZE + 1)   /* Size of a pathname */

BYTE Filename[FCHARS + 1];
BYTE Directory[MAXPATH] = "DF0:";

struct FileRequester FR = {
     "Greetings! Click on stuff!",  /* Hailing text                  */
     Filename,      /* filename array                               */
     Directory,     /* directory array                              */
     NULL,          /* Window, NULL == workbench                    */
     NULL,          /* Flags, not used in this release              */
     NULL,          /* Function pointers, not used in this release */
     NULL,          /*     "          "          "                  */
};

main()
{
     char *Selection;

     if ( Selection = FileRequest( &FR ) )
     {
          if ( *Selection )  /* Check for valid filename */
               Printf("Filename = %s ", Filename);
          else
               Printf("No Filename selected, ");
          Printf("Directory = %s\n", Directory);
     }
     else
          Puts("User cancelled Requester!");
}
```

Listing 4, "Devices.c": Using ARP's AddDADevs() function to display devices in the system (disks and/or volumes and/or assigns)

```
/* Devices.c -- arp.library demo, uses DA lists to determine what devices
 *    are connected to the system. -=+SDB+=-
 *
 * Copyright (c) 1987, Scott Ballantyne
 * Use and abuse as you please.
 *
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <arpfunctions.h>

struct DirectoryEntry *DeviceList = NULL; /* List header */

#define ARG_DEVICES  1
#define ARG_DISKS    2
#define ARG_VOLUMES  3
#define ARG_ASSIGNS  4

char *CLI_Template = "DEVICES/S,DISKS/S,VOLUMES/S,ASSIGNS/S";
/* Example of a more extended help message */
/* It would be pleasant if more programs extended this sort of help */

char *CLI_Help =
"Displays devices connected to this system.\nYou may enter:\n\
\tDEVICES   -- To display all devices (this is default)\n\
\tDISKS     -- To display all diskbased devices\n\
\tVOLUMES   -- To display all volumes\n\
\tASSIGNS   -- To display all logical assignments\n\n\
Or you may enter any combination of the above.  Note that\n\
DEVICES DISKS is the same as simply DISKS\n";

main(argc, argv)
int argc;
char **argv;
{
     LONGBITS WhichDevices = 0;
     LONG   numdevs;
     register struct DirectoryEntry *dev;

     if (argc < 2 ) /* set default */
          WhichDevices |= DLF_DEVICES;

     /* Find out which switches were used, if any */

     if ( argv[ARG_DEVICES] )
          WhichDevices |= DLF_DEVICES;
     if ( argv[ARG_DISKS] )
          WhichDevices |= ( DLF_DEVICES | DLF_DISKONLY );
     if ( argv[ARG_VOLUMES] )
          WhichDevices |= DLF_VOLUMES;
     if ( argv[ARG_ASSIGNS] )
          WhichDevices |= DLF_DIRS;

     /* Now get the requested devices */

     if ( (numdevs = AddDADevs( &DeviceList, WhichDevices )) == 0 )
     {
          Puts("No entries found");
          exit(0);
     }
     Printf("Found %ld entries\n", numdevs);
     /* Now display the name and the type of device it is. */

     for ( dev = DeviceList; dev; dev = dev->de_Next)
     {
          Printf("%-32s", dev->de_Name);
          switch (dev->de_Type)
          {
               case DLX_DEVICE:
                    Puts("Resident device");
                    break;
               case DLX_VOLUME:
                    Puts("Volume [mounted]");
                    break;
               case DLX_UNMOUNTED:
                    Puts("Volume [not mounted]");
                    break;
               case DLX_ASSIGN:
                    Puts("Logical Assignment");
                    break;
               default:
                    Puts("Encountered Device of mystery!");
          }
     }
     FreeDAList(DeviceList);
}
```

Listing 5, "WildDemo1.c": Using ARP's wildcard pattern matching for filenames – see sidebar.

```
/*
 * WildDemo.c -- Demonstration of how to use the arp.library wildcard
 *          routines. This program simulates a sort of dir command.
 *
 * -+=SDB=+-
 *
 * Copyright (c) 1987, Scott Ballantyne
 * Use and abuse as you please.
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <libraries/dos.h> /* For ERROR_NO_MORE_ENTRIES, ^C, etc. */
#include <arpfunctions.h>

/* A slightly extended AnchorPath structure */
/* We extend the anchor structure in this way because we will be
 * using the pattern matching functions to build complete path names.
 * If you don't want to use this ability, then you don't need to extend
 * the structure at all
 */

struct UserAnchor {
     struct  AnchorPath  ua_AP;
     BYTE    moremem[255];  /* cheap way to extend ap_Buf[] */
};

/* Our templates and help strings */

char *CLI_Template = "Pattern";
char *CLI_Help = "List Files or directories by wildcard";

main(argc, argv)
int argc;
char **argv;
{
     struct  UserAnchor *Anchor;
     char    *pat;
     LONG    Result;

     if (argc < 2)
          pat = "*"; /* All files/dirs in current directory */
     else
          pat = argv[1];

     /* Because a struct AnchorPath contains the BCPL structure
      * FileInfoBlock, we must always insure that the AnchorPath is
      * longword aligned.  A simple way to do this and also track it
      * is to call ArpAlloc()
      */

     if ( Anchor = (struct UserAnchor *)ArpAlloc( (ULONG)sizeof( *Anchor )) )
     {
          Anchor->ua_AP.ap_Length = 255; /* Want full path built */
```

```
        /* We will be breaking on control 'C's */
        /* and on control 'D's, just to show how */
        Anchor->ua_AP.ap_BreakBits |= SIGBREAKF_CTRL_C;
        Anchor->ua_AP.ap_BreakBits |= SIGBREAKF_CTRL_D;
    }
    else
    {
        Puts("No memory!");
        exit(20);
    }

    Result = FindFirst( pat, Anchor);

    while ( Result == 0 )
    {
        Printf("%s", Anchor->ua_AP.ap_Buf); /* Display name */

        if (Anchor->ua_AP.ap_Info.fib_DirEntryType >= 0)
            Puts( "\t(dir)" );    /* Display type */
        else
            Puts("");  /* Just a newline */
        Result = FindNext( Anchor );
    }

    /* Free the Anchor chain built by the above functions */
    FreeAnchorChain( Anchor );

    /* Finally, check the result - if it is ERROR_NO_MORE_ENTRIES then
     * we are exiting normally, otherwise, it is either a ^C break
     * or an actual error of some kind.
     */
    if (Result == ERROR_BREAK) /* Control 'C' */
    {
        if ( Anchor->ua_AP.ap_FoundBreak & SIGBREAKF_CTRL_C)
            Puts("***Break on '^C'");
        else
            Puts("***Break on '^D'");
    }
    else if (Result == ERROR_OBJECT_NOT_FOUND)
    {
        Printf("Can't find %s\n", pat);
        exit(20);
    }
    else if (Result == ERROR_BUFFER_OVERFLOW)
    {
        Puts("I should have allocated a larger buffer. Sorry!");
        exit(20);
    }
    else if (Result != ERROR_NO_MORE_ENTRIES)
    {
        Puts("An IO error occured!");
        Printf("%ld\n", Result);
        exit(20);
    }
    /* else */
    exit(0);
}
```

Listing 6, 'WildDemo2.c': Second wildcard pattern matching example.

```
/*
 * WildDemo2.c - Search filesystem for patterns, and separate into directories
 *     and files, sorting each separately using DA lists.
 *
 * -+=SDB=+-
 *
 * Copyright (c) 1987, Scott Ballantyne
 * Use and abuse as you please.
 */

#include <exec/types.h>
#include <libraries/arpbase.h>
#include <libraries/dos.h>
#include <arpfunctions.h>

struct DirectoryEntry  *FileList = NULL;  /* Head of DAList */

/* Our secondary keys for sorting the list, we want the directories to bunch
 * up at the beginning.
 */

#define DIRECTORY  0L
#define FILE       1L

/* Our templates and help strings */
char *CLI_Template = "Pattern";
char *CLI_Help = "List Files or directories by wildcard";

BOOL lflag; /* Cosmetics */

main(argc, argv)
int argc;
char **argv;
{
    struct AnchorPath *Anchor;
    register struct DirectoryEntry  *de;  /* For later */
    register LONG fcount = 0, dcount = 0;
    LONG  key;
    char  *pat;
    LONG  Result;

    if (argc < 2)
        pat = "*"; /* All files/dirs in current directory */
    else
        pat = argv[1];

    /* Get our AnchorBase */

    if ( Anchor = (struct AnchorPath *)ArpAlloc( (ULONG)sizeof( *Anchor )) )
    {
        Anchor->ap_Length = 0;
        Anchor->ap_BreakBits |= SIGBREAKF_CTRL_C;
    }
    else
    {
        Puts("No memory!");
        exit(20);
    }

    Result = FindFirst( pat, Anchor);

    while ( Result == 0 )
    {

        if (Anchor->ap_Info.fib_DirEntryType >= 0)
        {
            key = DIRECTORY;
            dcount++;
        }
        else
        {
            key = FILE;
            fcount++;
        }

        if ( !AddDANode( Anchor->ap_Info.fib_FileName, &FileList, 0L, key))
        {
            Puts("Out of memory!");
            FreeDAList( FileList );
            exit(20);
        }
        Result = FindNext( Anchor );
    }

    /* Free the Anchor chain built by the above functions */
    FreeAnchorChain( Anchor );

    if (Result == ERROR_BREAK) /* Control 'C' */
        Puts("***Break");
    else if (Result == ERROR_OBJECT_NOT_FOUND)
    {
        Printf("Can't find %s\n", pat);
        exit(20);
    }
    else if (Result != ERROR_NO_MORE_ENTRIES)
    {
        Printf("ERROR #%ld\n", Result);
        exit(20);
    }

    /* Now munge through the DA list, displaying the sorted files */

    de = FileList;
    Printf("%ld Directories:", dcount);

    if (de->de_Type == DIRECTORY)  /* While we have directories */
    {
        lflag = 1;
        for ( ; de ; de = de->de_Next)
        {
            if (de->de_Type != DIRECTORY)
                break;
            columnize(de->de_Name);
        }
    }
    /* And now do the files */
    Printf("\n%ld Files:", fcount);
    if ( de )  /* Still more entries */
    {
        lflag = 1;
        for ( ; de ; de = de->de_Next)
            columnize(de->de_Name);
    }
    Puts("");
    FreeDAList( FileList );
}

/* Simple routine to print a file or directory name in 2 columns */

columnize(name)
char *name;
{
    if (lflag)
        Printf("\n\t");
    Printf("%-32s", name);
    lflag = !lflag;
}
```

Amiga Machine Language – A Detailed Example

by Jim Butterfield

© Copyright 1987 Jim Butterfield

A step by step tour of an upgraded ECHO command

Jim Butterfield is a Toronto writer, programmer and lecturer whose association with microcomputers goes back to the 1K KIM-1 in the earliest days of Commodore computing. Jim's ability to communicate his encyclopedic knowledge of CBM products through articles, books, lectures and television programs has made his name a household word among Commodore users around the world.

Transactor, Volume 8, Issue 3, contained two articles dealing with the Amiga's ECHO command. One was a program by Neal Bridges, giving a replacement ECHO command. The revised command was shorter and more flexible than the one provided by Commodore. . . but didn't support the "asterisk" (BCPL escape) mode outlined in my own article nearby.

So I made some small changes to the code to include this feature. The code is still mostly Neal's, but I've modified bits here and there. Check with the original article for more detail; a comparison of that article with this one will let you spot the changes I've made.

Neal wrote the program in a manner that makes it ideal for study purposes. It's completely self-contained, since it doesn't depend on external macros and doesn't need any support subroutines other than the standard ones in the Kernel.

So I'd like to take you through the code, explaining it as I go. I'll also need to explain a few things about how you tap the Amiga operating system. . . and that's something well worth learning, too.

To make it easier for you to read, I have taken out Neal's macro definitions and register equates. As his comments noted, these were to help reduce finger fatigue. I think you'll find the code more readable when it's shown in-line. To save space, I've removed most of the comment lines; you can reference the original article to see these. Keep in mind that Neal marked his original program as "Permission to copy but not to sell".

Line numbers are shown to allow the descriptive text to reference other parts of the program. You would not normally use them in writing code, although they would show up on your "assembly listing" output. Here we go.

```
2  ; Neal Bridges,
4  ; Welland, Ontario,
7  ;
8  ; A little extra chopping by Jim Butterfield.
```

Just some of the comments. Here come Neal's register equates. . . this listing won't use them, but you can see the intended use of some of the processor's *registers*.

```
57 bufptr      equr a2      ;Output buffer pointer.
58 argpstr     equr a3      ;Storage place for pointer
59                          ;to start of argument line.
60 argptr      equr a5      ;Ptr to start of
61                          ; argument line.
62 char        equr d0      ;Char being read in
63                          ; from argument line.
64 BitFlags    equr d1      ;Type of line termination.
65 SpecFlag    equr d2      ;Flag for special symbols.
66 count       equr d3      ;Length of output line.
67 arglen      equr d4      ;Length of argument line.
68 Dosbase     equr d5      ;Dos library pointer.
69 output      equr d7      ;CLI outhandle.
```

The above values are stored in *registers* within the processor, so they will be available much more quickly than if they were stored at memory addresses. If we had values in the registers when we called a subroutine, we might need to save these values, since the subroutine could change them.

Next comes an important address. It's the only "fixed" address within the Amiga – address 4. Contained in address 4 (actually, in addresses 4 to 7 inclusive) is a pointer to the main memory map (called the "Exec Library") within the Amiga.

```
72 AbsExecBase  equ $4
```

The pointer contained in address 4 will *not* necessarily be the same on different Amigas. If you have internal expansion RAM, the "Exec Library" will be placed higher in memory than it would under other configurations. So you *must* peek at address 4 to see where the "Exec Library" is located.

Now we supply some constants that we'll be using in the program: LF for the linefeed character; CTRLmask as a mask to generate "control" characters; and CASEmask to make upper and lower case characters look the same, if desired.

```
74 LF          equ 10          ;Linefeed value.
75 CTRLmask    equ %00011111 ;Mask to make CTRL chars.
76 CASEmask    equ %01011111 ;Mask to kill lower case.
```

Next come the "external references" – values to be taken out of external tables. All of the "xref" values given here could be simply defined as numeric values; they are the same in all Amigas.

```
78             xref _LVOClose
79             xref _LVOOpenLibrary
80             xref _LVOCloseLibrary
81             xref _LVOOutput
82             xref _LVOWrite
83             xref _LVOInput
84             xref _LVORead
```

An interesting thing about the above values: Ignore the "_LVO" part for the moment (it stands for Library Vector Offset, or something like that). You might suspect from the names (Close, Open, etc.) that these are addresses of Kernel subroutines; and you'd be almost right. They are not quite addresses; instead, they are "offsets" in a *jump table*. We don't know where the jump table itself is, yet. . . things seem to slide around in the Amiga. But when we find it, we'll use these offset values (which happen to be negative) to get to the proper jump. We'll talk about these offsets later.

We're ready to get on with the job. When our program starts, the system will set up two important values for us. Both values tell us about the command line.

Suppose, for example, we had commanded **ECHO DING DONG**. The first word, ECHO, causes this program, ECHO, to be loaded. The system then takes the remainder of the line, DING DONG, and places it in storage. The *address* of this text will be found in register A0; the *length* will be found in register D0.

The two values in A0 and D0 are important to us, of course, so we'll copy them into other registers (A5/A3 and D4).

```
86 ;-- Initial setup:
87 _main        move.l a0,a5
88              move.l a0,a3
89              move.l d0,d4
```

The *move* command, above, is almost self-explanatory. The tag ".l" tells us it's a "long" move – we move the whole 32 bits from one register to another.

Now, we'll need to work with two *libraries*. . . two important parts of the Amiga Kernel. One library is *Exec*; we've already mentioned it, and we'll find a pointer to Exec in address 4. The other library is *dos*. That stands for "disk operating system", but it handles much more than disks; it's needed for all input and output, among other things.

You'll find a lot of details about DOS in *The AmigaDOS Manual* (Bantam). The section of that book we're interested in is called "AmigaDOS Developer's Manual" (starting on page 161 in my edition).

So we can find the Exec library easily by checking location 4; but how do we find the *DOS library*? It's not hard: we ask the Exec library to find it for us. We supply the name, and Exec goes hunting for it.

We define the name, "dos.library", near the end of this program – you'll find it at line 235, below. It's a string, of course: a character "d" followed by an "o" and so forth. And we must use a binary zero to signal the end of the string. By the way: upper and lower case is *very* important when supplying a library name. Put a capital letter in the wrong place, and you won't find the library you're looking for.

So we take the library name, and put its address into register A1. (Command LEA below means "load effective address", and the PC reference means we're using *relative* addressing). We're not fussy about the DOS library version, so we put a zero in D0 as the version number; CLR.L D0 means "Clear D0 to zero".

```
90              lea dosname(pc),a1
91              clr.l d0
```

So we're going to ask the Exec library to find the DOS library for us. Let's get the pointer to *Exec*.

```
93              move.l AbsExecBase,a6
```

Now the address of Exec is safely in register A6 (AbsExecBase is address 4, remember?), and we're ready to call the routine OpenLibrary, which is an *Exec* routine.

OpenLibrary is quite a clever routine. It goes looking for the named library. If it doesn't find it in memory, it may fetch it from disk. In the case of DOS, it will certainly be in memory. Then it "nails down" the library to ensure that it won't move or disappear while you are using it. Later, CloseLibrary frees the library if nobody else needs it.

So A6 contains a reference pointer to *Exec*. The OpenLibrary jump is located below this pointer; 228 hexadecimal locations below, to be exact. Figure 1 shows how the table *Exec* is organized. We don't need to know the value of –$228, of course; our xref (external reference) above got the correct number for us. But if you like to see the "inner workings", it's useful to know that the following command (JSR for jump subroutine) could be written as JSR –$228(A6). That means: take the address in A6, subtract $228 from it, and then Jump Subroutine (JSR) to the resulting address.

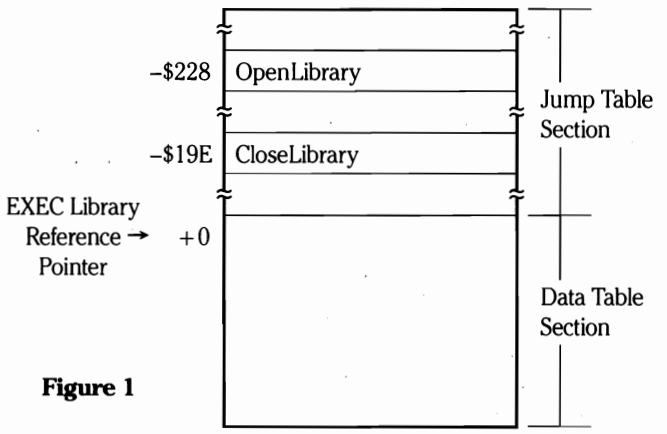


Figure 1

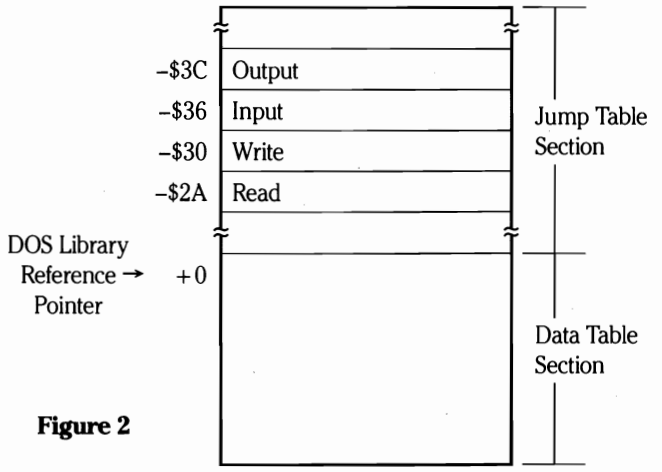


Figure 2

The OpenLibrary subroutine will, if it works correctly, give us a pointer to the DOS library. If it cannot do that for some reason, it will give us a zero value.

```
94              jsr _LVOOpenLibrary(a6)
```

The pointer to DOS – or a zero – will be given back to us in register D0. Let's move it to a safe place (register D5, say), and check to see if it's zero. If so, we can't do much, and might as well quit.

```
95              move.l d0,d5
96              beq close
```

Assuming we didn't take the above branch, register D5 now contains a pointer to the DOS library. You'll see a diagram of this library in figure 2. I've marked the subroutines that we will be using.

In case you're wondering: the subroutine OpenLibrary did not disturb the contents of registers A3, A5, A6, or D4. The rule is this (and if you write a subroutine, keep it in mind): subroutines are allowed to disturb the contents of A0, A1, D0 and D1; all other registers must be preserved. And don't even *think* of using register A7 unless you know all the rules about stack pointers.

One of the reasons we wanted the DOS library is to do input and output. We want to link to the existing CLI window, so we must ask DOS to give us an address (a *handle*) that will link output to the current window. We do this by calling *Output*. If you have the AmigaDOS manual, check the description of this routine. You'll see it needs no calling values, and will return the "handle" in register D0.

The pointer to the DOS library is in register D5; we'll move it to an address register (A6), and then call Output using the proper offset. The resulting "handle" will be copied to D7.

```
100 getout      move.l d5,a6
101             jsr _LVOOutput(a6)
102             move.l d0,d7
```

We're nicely set up, and ready to look at our data. First, let's make sure that we have *anything* following our ECHO command.

The number of characters is in D4; we'll check to see if it's less than or equal to 1. If so, give up – a single linefeed character would be equivalent to "no parameters". CMPI is "compare immediate", and we'll use ".b" to compare only a single *byte* of register D4.

```
105 checklen    cmpi.b #1,d4
106             ble close
```

Okay, we have something following ECHO. If it's a question mark followed by a linefeed, we need to do a prompt. Let's see if that's the case. Remember that this time we will compare two bytes – a *word* – with one comparison.

```
109 checkqm     cmpi.w #'?'*256+LF,(a5)
110             bne.s strtscan
```

Normally, we'll skip ahead. But if a prompt is needed, the following code will do the job.

We must print the "template" – in this case, a colon character followed by a space – that is stored near the end of this program, at line 237. The DOS subroutine *Write* will do the job. We'll set up the string address in D2 (with a side trip via A2), and the "handle" in D1, and the number of characters in D3. Again, check with your DOS Manual if you have it.

```
116 getline     lea template(pc),a2
117             move.l a2,d2
118             move.l d7,d1
119             moveq.l #endtemp–template,d3
120             move.l d5,a6
121             jsr _LVOWrite(a6)
```

We've sent the prompt. Now we must get the user input. As with output, we must first obtain an *input handle* (DOS subroutine *Input*).

```
124             jsr _LVOInput(a6)
126             move.l d0,d1
```

We have the handle. Let's read the line, using *Read*. D1 is the input handle, D2 is the buffer location (same place as the previous input), and D3 is permissible length.

```
127             move.l a5,d2
128             move.l #255,d3
129             jsr _LVORead(a6)
```

Now we have the input, we'll go back and check it again. And we need to move the length (returned to us in D0) to D4 where it belongs.

```
130             move.l d0,d4
131             bra.s checklen
```

The mainstream code comes in here. We're ready to look at the line and prepare for output. First, let's clear the flags that signal "quotes" and end-of-input. These flags are in D1. And we'll set up the output pointer in register A2.

```
135 strtscan    clr.b d1
137             move.l a5,a2
```

At the start of our input, we'll look for the "–n" sequence.

If we find it, we'll note the circumstances in register D1 and scan ahead to the first non-space.

```
145             cmpi.w #'–n',(a5)+
146             bne.s fixptr
147             moveq #1,d1
```

Again, note that we compare two bytes at a time (".w" for word). The compare instruction also advances pointer A5 automatically (that's what the "+" does). Now, we'll scan past the spaces:

```
153 findspc     cmpi.b #' ',(a5)+
154             ble.s findspc
155             subq.l #1,a5
156             bra.s setup
```

When we find a non-space, we back up one position (that's what the SUB, subtract, does), and then skip ahead to the main show. If we had *not* found the "–n" sequence, the code would come to the next instruction. At this point, we'd need to back up the pointer.

```
159 fixptr      move.l a3,a5
```

And the "–n" code joins here. As we start the main part of the job, we set our output-count (D3) to zero, and clear our special flag register (D2).

```
161 setup       clr.l d3
162             clr.b d2
```

Our main loop comes back here. We grab a character, and check it to see if it's any of a number of special characters. If it's a linefeed, for example, we're finished and can proceed to the output routine.

```
166 scan        move.b (a5)+,d0
168             cmpi.b #LF,d0
169             beq exit
```

Now, if the "special characters flag" (D2) is not zero, we have extra work to do. Normally, it will be zero, and we'll zoom ahead to NOFLAG.

```
174             cmpi.b #1,d2
175             blt.s noflag
```

Well, if we got here, the flag was set. The previous character was either an asterisk or an up-arrow. We can check which one very quickly, since the compare flags above are still valid.

```
180             bne.s ctrldo
```

If the previous character was an up-arrow, we've hopped ahead to CTRLDO. The code here will be for the character following an asterisk. We must see if the following character is an E or N (upper or lower case), in which case we must change it to Escape (code 27) or NewLine (code 10). We'll use register D2 for testing the character.

```
182             move.b d0,d2
183             andi.b #CASEmask,d2
184             cmpi.b #69,d2
```

If we find an E, we'll substitute the ESC character.

```
185             bne.s try2
186             move.b #27,d0
188 try2        cmpi.b #78,d2
```

If we find an N, we'll substitute NewLine.

```
189             bne.s  putinbuf
190             move.b #10,d0
```

The next code handles the character following an up-arrow. It doesn't matter that the asterisk-N (NewLine) sequence also falls in here. . . it won't come to any harm.

```
192 ctrldo      andi.b #CTRLmask,d0
```

The normal character handling is done in the following code.

(Again, it doesn't matter that up-arrow sequences fall in here). If we find a quotes character, we throw it away.

```
197 noflag      cmpi.b #'"',d0
198             beq.s scan
```

Now we look to see if the character just received is one of the special ones – asterisk or up-arrow. If so, we'll note it in register D2.

```
201 chkspcl     moveq #1,d2
202             cmpi.b #'*',d0
203             beq.s scan
205             moveq #2,d2
206 chkother    cmpi.b #'^',d0
207             beq.s scan
```

Note the "backwards" logic used above. We set D2 first, perform the test, and then reset D2 if it doesn't pass the test. The final D2 setup follows; and at this point, we deliver the desired character to the output buffer, adding one to the output count in D3. After that, back to get the next character.

```
210 putinbuf    clr.b d2
211             move.b d0,(a2)+
213             addq.b #1,d3
214             bra.s scan
```

When the input has been completely scanned, it's time to do the output. We check the D1 flag to see whether or not to add a NewLine character to the end of the output.

```
218 exit        tst.b d1
219             bne.s nolf
221             move.b #LF,(a2)
222             addq.b #1,d3
```

Linefeed or not, we're ready to do the output. Again, we must set things up for our call to *Write*. (See lines 116 to 121, above).

```
223 nolf        move.l a3,d2
224             move.l d7,d1
```

Register D3 already contains the number of characters to be written.

```
226             jsr _LVOWrite(a6)
```

It's all over. Release the libraries and quit.

```
229 close       move.l d5,a1
230             move.l AbsExecBase,a6
231             jsr _LVOCloseLibrary(a6)
232 close1      rts
```

As promised, here's the string containing the name of the DOS library. Remember that upper/lower case is important for this kind of usage.

```
235 dosname     dc.b 'dos.library',0
```

The library string, above, needed to be followed by a zero byte. Not so for the template string (colon, space), whose size is defined separately. That's why we define label ENDTEMP – to note the end of the string so that we can subtract it from the start of the string and get the size.

```
237 template    dc.b ': '
238 endtemp
```

And here's the end of our assembly source:

```
240   end
```

Not as simple as on the eight-bitters. . . but still quite manageable.

. . . And What Are Little Fonts Made Of?

by Betty Clay

Everything you ever wanted to know about font definition files

Betty Clay is a mathematics teacher ("Retired at last!" she says) by profession, and a student of Commodore computers by avocation. Betty can be reached via Compuserve (74145,657), QuantumLink (bjc), or by mail at 1322 South Oak Street, Arlington, Texas, 76010.

If you examine the FONTS: directory on your Workbench disk, you will see a number of directory names (one for each of the disk fonts, such as Opal and Garnet) and, below the directory names, a group of files bearing the same names, but with the extension ".font".

Going further, if you look at one of the individual directories (the Garnet directory, say), you will find two files with the unusual names of '9' and '16'.

What are all these files, and what do they mean? Just how is a font structured on the Amiga?

The .font Files

These files are actually font header files, and they contain only a description of the font. If there is only one size or style of the font a font header contains exactly 264 bytes. You can get a "hex dump" of this file from your Workbench by using the command:

type fonts/garnet.font opt h

and here is part of what you will see:

```
0F000002 6761726E 65742F31 36000000
```

followed by a large number of zeroes. This can be interpreted rather easily:

0F00	indicates that this is indeed a font header file.
0002	means that there are two definitions (sizes or styles) for this font

The next part of the file describes the first of the Garnet fonts, Garnet 16.

6761726E 6574	the ASCII representation of "garnet"
2F31 36000000	the 2F represents the '/', and 3136 is the ASCII representation of 16, the height in pixels of this font.

Then there will be 244 bytes of zeroes, and at the end of the description will be these items:

0010	$10 = 16, the pixel height of the font
00	the style of the font. The font could be designated as being of a particular style, such as underlined, bold, or italic. If that were the case, which it isn't here, bits would be set in this byte to reflect the style. When the style is built into the font in this way, it cannot be changed with (for example) the menu in Notepad.
62	the flags set for this font. In this case, the 6 indicates that it is a "designed" rather than a "constructed" font, and that it is proportional rather than fixed-width, and the 2 indicates that it is a disk-based font. For a complete list of flag definitions, see the header file "graphics/text.h".

In the "Garnet.font" example, these data are immediately followed by a description of the other file, "Garnet/9". It is exactly like the one described above with a single exception: the second-from-last byte is 0009, which is the Ysize, or pixel height, of Garnet/9.

There will be a similar description, each containing the same information, for each size or style in this 'family' of fonts. The "fontname.font" file can grow quite long if there are many sizes in the family.

Oh, yes – all those zeroes? They are just place holders, put there to allow you all the room you need (up to 256 bytes) to tell AmigaDOS the entire directory path it must follow in order to locate your font.

Those Files With Numbers For Names. . .

Still using the Garnet font for an example, the next thing to examine is the subdirectory named "Garnet". To reach it, ask for the directory of "Fonts:Garnet". There you will find two files, named '16' and '9'.

The definition of the font itself is held in these files that are named only with a font-height in pixels. The files can be extremely complex. In the case of a fixed-width font, far less information is needed, but a proportional font requires four separate tables to hold all the information needed to describe it fully.

The first part of any font file definition contains a node structure, which permits the font to be linked with the rest of the system, a space where the name of the font will be entered after it is loaded, and a "TextFont" data structure containing information needed by the system to manage the font. For example, the TextFont structure contains a counter that keeps up with the number of tasks that are using the font, and that will change while the font is held in memory and different tasks are accessing it. You will not be permitted to remove a font if another task is using it, nor can you change the "Assign fonts:" if some task is still using one of the fonts in the current assignment (that is, until this flag has been decremented to zero). Instead, you will receive an error message, "Can't cancel fonts".

Here is part of a hex dump of the actual "Garnet/16" file. You can get the entire hex dump from your own Workbench disk with the command:

```
type fonts/garnet/16 opt h
```

I have used just enough of the file to show its meaning. You may wish to look at the entire file. By studying the file we can see exactly how it is organized:

```
0000: 000003F3 00000000 00000001 00000000
0010: 00000000 0000055E
```

This is loading information, necessary because a font definition is just a loadable file to the Amiga. This part is a "hunk_header" block, indicated by the number $000003F3. The second long-word indicates that no resident libraries have to be opened when the file is loaded, and the third that only one hunk must be loaded. Then there are two long-words that specify the positions of the first and last hunks of the file in a hunk table used by the system loader. Because there is only one hunk in this case, both these long words have the value $00000000. The last number in the block is the size of the hunk in long-words. This is the only item in the header block that will vary from one font file to another.

```
0010: 00000000 0000055E 000003E9 0000055E
```

The last two long-words here indicate that this is a "hunk_code" ($3E9), as opposed to "hunk_data" or "hunk_bss" (uninitialized data), and again gives the length in long words, $55E.

```
0020: 70004E75 00000000 00000000 0C000000
```

In this line, the first word, $7000, is the assembly language instruction MOVE.Q #0,D0; and the second word, $4E75, is the instruction RTS. These instructions are placed here for safety, in case someone tries to execute this file instead of accessing it properly with an Exec OpenDiskFont() call. Then begins the node structure, which will be used to link the font into the system's list of loaded disk fonts, which originates in the DiskfontBase Library structure (the Diskfont Library must be opened by any program that wants to use disk-based fonts). The first two long words in the node structure are pointers to the next and the preceding node in the list; these pointers are null here since the font isn't loaded. Then comes the node type (0C = NT_FONT), priority (00), and a pointer to the node name.

The value of this pointer ($0000001A) is actually an offset from the start of the hunk at 0020 above. When the font is brought into memory, this pointer's value will be adjusted by the system loader to reflect the font's actual load address, which cannot be known ahead of time. You can see here that the data pointed to (at 0020 + 001A = 003A) is null; in fact, it consists of 32 consecutive zeroes. This is filled in by the OpenDiskFont routine when the font is loaded.

After the pointer to the node name comes a font file identifier (0F80), a revision number (0001), and an empty long-word slot for DOS to store the font's segment address after loading. This brings us to 003A, and the 32 bytes for storing the font name.

At 005A follows another node structure identical to the one we have just finished discussing. This node, which will be used to link the font into the system's list of all active fonts (not just disk-based fonts) is part of an Exec message structure, which is in turn the first member of the DiskFont structure for this font. When the font is closed by its last accessor, and is thus no longer needed by any program, the system will reply to the message port pointed at in the message as a signal that the font can be unloaded.

The last item in the message structure, the message size, is at 006C. This size is $1508 bytes, which encompasses all the font data proper – the character shape table and its three associated tables. If you're thinking about constructing your own font files, note that the example fonts program in the ROM Kernal Manual sets this message size incorrectly with the line:

```
DC.W    fontEnd-font        ;mn_Length
```

The correct version would be:

```
DC.W    fontEnd-fontData    ;mn_Length
```

```
0030: 001A0F80 00010000 00000000 00000000
0040: 00000000 00000000 00000000 00000000
0050: 00000000 00000000 00000000 00000000
0060: 00000C00 0000001A 00000000 15080010
```

The part of the file that is most meaningful for people begins at byte $006E. This is where the real font definition begins, with the balance of the TextFont structure – the part after the Exec message – as documented in the include file graphics/text.h (or graphics/text.i for the assembly language definition). The structure begins with the font's Y-size – its height in pixels:

```
0060: 00000C00 0000001A 00000000 15080010
```

Garnet/16 has a Y-size (height) of $10 = 16 pixels.

```
0070: 00620014 000B0001 000020FF 0000006E
```

The style bits are $00 (0 for normal font) and the $62, the flag bits, indicate that this is a "designed" font, that it is proportional, and that it is disk-based.

```
0070: 00620014 000B0001 000020FF 0000006E
```

The X-size or width of the font is $14 (decimal 20). This is the nominal width, but since we are studying a proportional font, each character will have its own width set by a spacing table.

```
0070: 00620014 000B0001 000020FF 0000006E
```

The baseline of the font is $000B – 11 lines from the top. Every character is placed on the screen relative to this line, which should be the bottom line of characters that do not have descenders.

```
0070: 00620014 000B0001 000020FF 0000006E
```

The font will be drawn with bold smear, which makes the font appear darker.

```
0070: 00620014 000B0001 000020FF 0000006E
```

This space is for the "accessors." When the font is loaded into memory, this is where the Amiga keeps a record of the number of tasks using this font. It is incremented and decremented as different tasks open and close the font.

```
0070: 00620014 000B0001 000020FF 0000006E
```

This font begins with the space character, whose ASCII value is $20 (decimal 32) and the last character has an ASCII value of $FF, decimal 255. This character is a member of the alternate-character set. Many fonts do not include this many characters. Most begin with the space and end after the lower-case 'z', or perhaps four characters later after the '~', which has the ASCII value of $7E, or decimal 126.

```
0070: 00620014 000B0001 000020FF 0000006E
```

The bitmap information for this font begins at offset $6E + $20 = $8E (see below).

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

This is the Modulo – the number of bytes the pointer must move to go from one line of the character to the next row of it. In this case, it would be $00E0 (240) bytes into the bit-packed table for each row. There is a more detailed explanation of this later in the article.

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

This is a pointer called the CharLoc, and it points to a table that tells how far into the bit-packed table the data for a given character begins. In this case, the CharLoc table will begin $E6E (3694) bytes from the beginning of the code hunk at 0020.

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

Here is the location of the CharSpace table, which tells how many spaces each character should have. If the font is not proportional, this table would not be needed – but most are proportional! For this proportional font, we should find this table at an offset of $11F2 (4594) bytes.

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

The Kerning table for Garnet/16 will be found at an offset of $13B4 (5044) bytes. For the Amiga, "kerning" is given a non-standard definition, telling how many pixels to leave blank before beginning the character. In orthodox usage, kerning indicates leftward movement of letters such as V or W when they are printed next to a letter with a parallel slant, such as A, making the text more attractive.

```
0080: 00E00000 0E6E0000 11F20000 13B40440
```

This is the beginning of the actual bitmap that describes the way the characters are to be drawn.

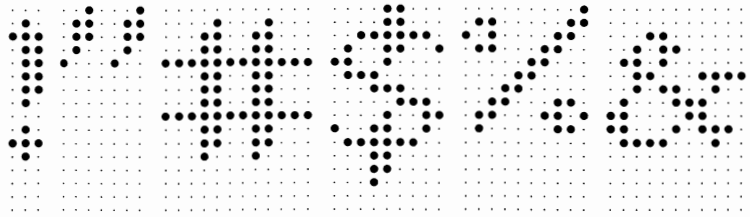
So What Is Bit-Packing?

The character definitions for Amiga fonts are encoded in such a way that the bits needed for drawing the characters (but not the spaces between them) are held in a table with instructions for the top line of the entire font first, then all of the second line, then the third, and so on. In the case of Garnet/16, the first character is a space that is eight pixels wide. Since there is a separate table for spaces, this character is omitted from the bitmap, and the next character, the '!' begins the table. Here is a bitmap picture of the first line of the first six characters:

```
00000100010000000000000000110000000000000100000000000...(more)
```

After all of the top lines of the entire font are written, the second row begins. To make these characters more apparent, they are indicated below with a '·' for a zero, and a '●' for a 1. The spaces between the characters are not in the file, but make the characters more readable in this form.

```
000 0010001 000000000000 000011000 0000000001 00000000000
```



Look carefully, and you will be able to see that these represent the '!', double quote, '#', '$', '%' and '&' characters – those with ASCII values of $21, $22, $23, $24, $25 and $26.

Now, change these binary digits to hexadecimal, remembering that to do so, you can just combine four binary digits to make one hexadecimal digit, with each column containing a one representing 8, 4, 2 or 1:

```
% 0000 0100 0100 0000 0000 0000 0011 0000 0000 0000 1000 0000 0000
```

means:

```
$ 0   4   4   0   0   0   3   0   0   0   8   0   0
```

Thus our bit-packed information can be translated to hexadecimal format, and the beginning of each of the rows of the font can be represented like this:

```
Row  1: $0440003000800
Row  2: $4CC22030C1800
Row  3: $ECC660FD430C0
Row  4: $68866132C6160
Row  5: $713FFF200C340
Row  6: $600661C01818F
Row  7: $6006607030164
Row  8: $4006601C63238
Row  9: $003FFC26C6E18
Row 10: $400662648362E
Row 11: $E00661F8003C4
Row 12: $4004406000000
Row 13: $0000006000000
Row 14: $0000004000000
```

Rows 15 and 16 are all zeroes. By using the Modulo number pointed out above, we know that Row 2 begins $E0, or 240 bytes after the beginning of row 1, and row 3 will begin 240 bytes after the beginning of row 2, and so on. Now we check it out in the hex dump:

```
0080: 00E00000 0E6E0000 11F20000 13B40440

0090: 00300080 04000000 00000004 00000000
```

Here are the hexadecimal representations of the top line of the six characters we drew. The remainder of the top line goes on until we reach byte $016D, 240 bytes further into the file.

```
0160: 00000000 00000000 00000000 00004CC2

0170: 2030C180 0E3F0000 0000000C 00000000
```

Here, beginning at $016E, we find the characters we had expected to find at the beginning of the second line from the top of the font. Going yet 240 bytes further, to $024E, we find:

```
0240:  00000000 1C007060 0001C000 0000ECC6

0250:  60FDA30C 06418000 00000018 00000000
```

(the same digits as in our table above). So now we see how the various rows of the actual instructions for drawing the font are stored in the file. After each 240 bytes, a new line begins until all sixteen lines (the Y-size) have been stored.

There is no need to reprint the entire file here, but you can type it on your own screen if you wish (TYPE fonts:garnet/16 opt h).

For our purpose, we skip now to $09E0, to find the CharLoc table. By looking at Garnet/16 in a font editor, we can see that the six characters at the beginning of the font occupy, in order, 3 bits, 7 bits, 12 bits, 9 bits, 10 bits and 11 bits. Here is the CharLoc table, and in it we find the words paired to show where in the table each character begins and how many bits wide each one is:

```
0E90:  00000000 00030003 0007000A 000C0016 ...
```

0000 0003 shows that the first character begins at the zero position and is three bits wide. 0003 0007 indicates that the second character begins at the third bit, and is seven bits wide. 000A 000C shows that the third character begins at bit 10 (3 + 7) and is $C or 12 bits wide. And the table goes on to describe up to 255 characters in the font, but in this case 255 – 32, 223.

But Spacing Is Important!

And so it is. The Amiga provides another table in which the space allowed for each character is recorded. In the section before the font description actually started, we found the "CharSpace" pointer. Now we look through the font to find the actual table in which those spaces are recorded. By checking the font in a font editor, we see that the number of pixel-width spaces allotted was eight for the space character itself, four for the '!', eight for the double quote, thirteen for the '#', seven for the '$', eleven for the '%', and twelve for the ampersand '&'. Looking down to $1212, we find this table:

```
1210:  00000008 00040008 000D0007 000B000C

1220:  00050007 0007000A 00080005 00090005
```

These are the hexadecimal representations of the exact number of spaces expected, and the table goes on to show the width of all the characters defined for this font.

But Isn't A Kern Some Kind Of Bird?

The typography text I have says that a kern is the part of a letter that overhangs the body of the type. In the Amiga, however, the kern represents the number of bits to be inserted before the character is drawn. So before any character is drawn, the text drawing routine looks to see how many bits to insert before drawing, and only then finds the character in the bit-table and draws it.

There was a pointer at the beginning of this file to help the Amiga find the kerning table. It was the last item before the actual bit-map of the characters began. In the C language definition of the TextFont structure, this pointer is called "tf_CharKern".

In a font editor we can observe that the six characters at the beginning of Garnet/16 have kerning values of zero for the space, one for the '!', zero for the double quote, zero for the '#', one for the '$', and zeroes for the '%' and the '&'.

Also, since it takes two bytes to indicate the spacing for each character, we must move down the file 223 * 2 , or 446 bytes to find the kerning table in line $13D0, beginning at the third byte:

```
13D0:  00090000 00000001 00000000 00010000

13E0:  00000002 00010001 00000001 00010001
```

The first two words, at $13D0 and $13D2, complete the spacing table, with nine bits for character $FF (255) and 0 bits for the dummy character that is to be printed if a user asks for a character not in this font.

At $13D2 we find the kerning table, beginning with zero kerning bits for the space, one for the '!', zero for the double quote, zero for the '#', one for the '$', zero for the '%', and zero for the '&'. The next character has two kerning bits, and so on through the entire font.

When the system displays a character, then, it must go through something like the following process:

1) Given the character number (e.g. 65 for 'A'), look in the kern table for the number of blank pixels to leave before displaying the character.

2) Now look in the CharLoc table for an index into the font bitmap for the top row of pixels for that character, and for the number of pixels that must be fetched from each row (i.e. the maximum visible width of the character).

3) Transfer pixels from the bitmap onto the screen using the blitter (for each bit-plane in the screen).

4) Look in the spacing table for the total width of the character (visible portion plus the space on either side), and advance the text cursor horizontally by that much.

Hunk_end

That concludes the study. An Amiga font consists of a descriptor file called "Fontname.font", a directory with the same name, and one or more fonts named for their pixel-height. These numbered fonts contain a node structure, a message structure, a group of pointers to various parts of the file, the detailed description of each character in bit-mapped form, a CharLoc table, a spacing table, and a kerning table. This is followed by a short section of information for system bookkeeping, including the relocation table for the six relocatable references in the file (two references to the name, one apiece to each of the four tables), and ends with the required "hunk_end", $3F2.

To be even more specific – a font is a set of tables of zeroes and ones that tell the computer which pixels to turn on and which to leave off, and how far apart to put them. And *that* is what little fonts are made of.

Warrior Cycles: Programs Battle it Out in an On-Screen Arena

by Rico Mariani

. . .the Amiga is obviously the perfect candidate for games like these. . .

Rico Mariani has been a part-time programmer at Comspec Communications Inc. from the days of the 8K PET, and an Amiga hacker from the machine's early days. A Computer Science student in the University of Waterloo's co-op program, he currently spends his work terms at the Ontario Science Centre, writing software for interactive displays on the Amiga and other 68000-based systems.

Several years ago I heard about a game that was available for the Apple II called "Robot Wars", in which each player programs a robot and then the robots are turned loose to fight each other. Such a challenge seemed like the ultimate test of programming skill: your program in a battle to the death against another. A game with a similar flavour, described recently in *Scientific American*, is "Core Wars", in which each player codes a small program in a special machine language. These programs are then turned loose in the memory of a virtual machine where they do battle – each program trying to write on top of the other so as to make it stop functioning. The beauty of "Core Wars" is that the programs are so simple and easy to write, yet there are countless strategies, each with a playing personality of its own.

A multitasking machine like the Amiga is obviously the perfect candidate for games like these, especially with its great graphics to show the game in progress. Here, then, is my multi-program competitive game for Amiga owners: Warrior Cycles.

Warrior Cycles is a game based on "Light Cycles" from the movie Tron. Each player writes a program that will control the navigation of a Cycle in an arena whose size is unknown until the game starts. The rules of the contest are simple: as a Cycle moves, it leaves a deadly trail behind it. A Cycle is eliminated when it runs into another Cycle or part of the trail left by some Cycle (even its own). The game continues until only one Cycle remains. The game may end in a tie (with no winner) if the last two or more players are eliminated simultaneously.

Warrior Cycles consists of a main "server" program, and one or more "player" programs. The server creates the arena in which the games are played, which is an Intuition Screen. The players are separate programs, written by the (human) contestants in the language of their choice. The players control the Cycles in the arena by asking the server information about what's on the screen, and telling the server in what direction they wish to move on each turn. The server is run from the CLI first (using the RUN command), followed by each of the players. The server will wait until a specified number of players enter the game before it begins.

The arena displays the path of the Cycles as lines, each in a unique colour. The number of games won by each Cycle is displayed in the screen's title bar in the same colour as the Cycle (when a game is completed, a new one automatically begins). The games are fast-paced and exciting to watch, and you quickly see which the better player is after a few rounds by watching the scores. Meanwhile, back on the Workbench screen, each player has created a small window of its own, each in a different position so you can see them all at once. The window titles show the players' names, and within the window the colour of the player ("yellow", "blue", etc.) is displayed, along with a constantly updated display of wins, losses and draws. You can retire a player from the game by clicking the window close gadget, and new players may be entered at any time by just running them from the CLI (players cannot enter or leave in mid-game, but will wait until a game ends or a new one begins). The server also has a close gadget; when the server is killed, it tells all players that it's about to shut down, and does so after they've all replied to the message.

While the tournament is in progress, you can watch the game on the Cycle screen, and just push the screen to the back or slide it down to see who's playing and how they're doing. You can even play along – a player has been written that has no strategy of its own, but just moves in the direction indicated by a joystick. You can compete with the "robot" players this way, or test a player you've written.

The current version of Warrior Cycles supports up to six players; this is not an intrinsic limitation of the program, but exists because (1) a three-bitplane (eight colour) screen is used; and (2) the more players (programs) there are running at the same time, the less CPU time each player can get between turns, so the longer the turns have to take. Lots of time between turns means a boring tournament, so the number of players in the tournament was kept down for the benefit of human spectators. As it is, you can have fast-paced games and still give the players enough time to think between moves.

The server can be given three options when started: the number of players to wait for before the game begins, the amount of time between moves, and the number of games to play before ending. It is important to set the number of players to wait for so that all players that are to join the tournament have registered before a game begins. If the game begins before everyone is in, one or more players might have to sit out until the next game. The amount of time between moves is specified in milliseconds per player, meaning you won't have to vary this amount with different numbers of players in a tournament. This value is typically set to 20 for a fast game; a higher value means that the player programs can have more sophisticated playing strategies, since they'll have more time to think. More about tournament time settings later.

Here is a typical invocation of the server program, asking for a tournament of ten games involving three players, at twenty milliseconds per turn (the "Lightweight" class, see below):

```
run cycle p3 g10 t20
```

The defaults for p, g and t are 2, 0 (for ever) and 6 respectively. These defaults are suitable for experimenting with two very simple-minded players; ordinarily you will want to adjust the settings yourself. The invocation of the players would be simply:

```
run player1
run player2
run player3
```

When you "run cycle", the server immediately puts up its own screen, which you will have to push to the back so that you can call up the players from the CLI. The best approach is therefore to put the above commands in a script file so that you can set the whole tournament in motion with:

```
execute tourney.cmd
```

Technical Details

The server process sets up the display screen and keeps track of the position and heading of each of the players. Each player (a "client" of the server) that wishes to join the game does so by sending a message to a public port owned by the server (see the article "Messages, Ports and Signals" in *Transactor*, Volume 8, Issue 2, for information about ports and messages). The server then accepts the new player and assigns it a colour. The client sends further messages to tell the server what heading it would like to take; the server's replies inform the client that the move has been completed and if the client has been either eliminated or declared the winner. All of the timesharing between players is done automatically by the Amiga multitasking Executive. When you "run" the server and then "run" whatever players you wish to participate, they will find each other and start playing.

One benefit of this method of implementation is that the actual players can be written in whatever language the author feels most comfortable with. As long as the player program sends messages in the required format to the server, it will be updated and informed of its status regardless of what language was used.

In addition to the server program, there is also a set of convenient "interface" routines that hide all of the message passing details involved in communicating with the server. These are written in C but could easily be converted to Modula-2, assembler or other languages if desired.

Routines for Player Programs

What follows is a brief synopsis of each function. If you want to write a player program, you should look carefully at the implementation of each of these functions. Beside each function is a note saying whether or not a function results in a message being sent to the server; when a message send is involved, the action that is taken will not happen immediately.

status = Register(colour) **(sends a message to server)**
int *colour;
This function tries to locate the server and inform it that your player wishes to join the game. The integer referenced by the colour argument will be filled in with your colour number. If registration is unsuccessful then a non-zero status will be returned. You must successfully register before any of the other functions can be used.

Retire() **(sends a message to server)**
This function will inform the server that you will be leaving the game; it can only be called after a successful registration. If you wish to rejoin later you must register again.

result = Inquire(x,y) **(no message sent to server)**
int x, y;
This function asks if the cell at coordinates (x,y) is filled, returning TRUE if it is and FALSE if it isn't. All cells occupied by a Cycle's trail, and all those outside the arena, are considered filled. In other words, if a player runs into a cell that Inquire() returns true for, it will hit either a trail or a wall, and be killed.

result = Look(dir) **(no message sent to server)**
int dir;
This function returns TRUE if the square in the indicated direction is obstructed. The 'dir' parameter must be one of DIR_UP, DIR_RIGHT, DIR_DOWN, or DIR_LEFT as defined in the file "cycles.h". See GetInfo() below.

image = GetScreenMem() **(no message sent to server)**
This function returns a pointer to memory that describes the bitmap for the current arena. *You must not write to this memory* as it is shared by all running Cycles. Once you have a pointer to the arena image you can find out how big the arena is and what squares are currently occupied. You could use this to (for instance) find all active players by making a copy of the bitmap and then comparing this against the new bitmap after all the Cycles have moved. See cycles.h to find out what the exact layout of an IMAGE is.

GetInfo(x, y, dir) **(no message sent to server)**
int *x, *y, *dir;
This function fills in the integers pointed to by x, y and dir with your current x position, y position and heading respectively. The heading is one of DIR_UP, DIR_RIGHT, DIR_DOWN, DIR_LEFT. Two macros are defined in cycles.h for transforming directions; they are called TURN_RIGHT and TURN_LEFT. TURN_RIGHT(dir) returns the direction you get by turning right if you're facing dir. TURN_LEFT(dir) operates symmetrically.

status = Direction(dir) **(sends message to server)**
int dir;
This function informs the server that you wish your next move to be in the direction indicated by the 'dir' argument. This function will return after that move has been executed. Note that this is a real-time system – the server expects to be informed at regular intervals of your new direction. If you spend too long thinking about which way to go, you simply continue on your current heading – if there's a wall there, too bad. However, if you make up your mind too fast then the opposition will have more time to think while you're waiting for the server to move you, so be careful. . . not too quick, but not too slow either. The return status will be one of:

STATUS_NEED_NEW:	ready for your next move
STATUS_LOSER:	too bad, you just lost
STATUS_WINNER:	celebrate, you won this game
STATUS_REMOVED:	server has to abort, game over.

Note: never try to send a message to the server after you get STATUS_REMOVED.

status = Await() **(sends message to server)**
This function tells the server that you wish to join in the next game, and that it should inform you when it has begun. Call this after you have been declared WINNER or LOSER. If you don't wish to continue then call Retire() instead. The status return is as in the Direction function except that you'll never be a WINNER or LOSER right away (the server may conceivably have to abort, however).

Programming Support

In order to simplify the writing of players I have created a simple manager program that takes care of opening and registering and all those other niceties. All you have to do is create three functions and a name for your player and you'll have a working program. Put the following functions in a module, link with the manager code, and your player is ready to battle it out in the Warrior Cycle arena.

int strategy_init();

Define this to do any initialization required by your player. If it returns non-zero, the manager will abort, removing the player program from the system.

int strategy();

This is the meat of your program, where it will call the server routines described above to look around, and figure out which way to go. It must return one of DIR_UP, DIR_DOWN, DIR_RIGHT or DIR_LEFT. If it returns a negative value, the manager will abort.

int strategy_finish();

Define this to do any clean-up that your player requires, like freeing allocated memory.

You will also need to make the following declaration:

char player_name[];

This should be an array of characters containing the name of your player program.

The best way to see how these can be used in your own player program is to look at the some of the sample players that I have written.

Player#	Name	Personality
1:	Mr. Right	if there's a problem ahead he turns right
2:	Dizzy	always tries to turn right if he can
3:	Vertigo	prefers going up/down to going left/right
4:	Round the clock	looks in every direction, picks the clearest
5:	Keep'em flying	as above but prefers current direction
6:	Not so Dizzy	Looks ahead 3 moves and picks best direction
7:	Look B4 You Leap	Looks ahead 4 moves, likes to go straight
8:	Joystick	Plug a joystick into port 1 and drive

The first three are what I call feather-weights: they decide their move by looking at the squares immediately around them. They respond to the server quite quickly. The next two are light-weights: they look around a little in a certain fixed way. Players 6 and 7 are middle-weights: they inspect their entire neighbourhood before moving on. The final class (heavy-weight, which I have no example for) considers the position of the opponents in the strategy. This class distinction is important because the rate at which the server updates the Cycles is not fixed. If this rate is set too high, the middle-weights will not have enough time to do their thinking, and the light-weights (who make up their minds faster) will dominate. An even lower setting would cause the feather-weights to dominate. . .

I have used the following settings for various player types:

feather:	10 ms/player
light:	20 ms/player
medium:	50 ms/player
heavy:	200 ms/player

Writing a good heavy-weight could be quite a challenge.

Win valuable prizes!

Now it's up to you to write your own player program and mail it in to *Transactor for the Amiga* for the great Cycle playoff. Write it in whatever language you wish – simply create some code to mimic the functions in the play.c file; it all comes down to simple message sends. The only rule is that you are not allowed to affect the relative performance of any other program you are competing against. This includes such things as changing your task's priority or your opponent's, creating new tasks to do computation for you (thus effectively giving you more CPU time), or modifying any of the shared data structures (when in doubt, make a copy and use that). Only programs in the same class will compete against each other; however, you may enter a program in as many classes as you wish. The style of tournament will be decided based on the number of entrants. I'm hoping for a round-robin but, if this will take years to run, I'll have to settle for a simpler format.

May the best Cycle win!

Listing 1: The source files for the Warrior Cycles server system. All these modules should be compiled and linked to produce the server "cycles" program. If you are using Manx, you can use the makefile in Listing 4 to compile everything ("make cycle" to compile the cycle server program only without compiling any players). Note that the program will only compile properly if compiled with the 32 bit integer option. Lattice C users should have no problem compiling this, but it hasn't been tested with that compiler.

Listing 1a: "cycles.c" – the central part of the server, where messages from the players come in and are dealt with.

```
/*
 * main cycles program. . . wait for messages to come in and
 * do "the right thing"
 */
#include <intuition/intuition.h>
#include <exec/memory.h>
#include <devices/timer.h>
#include <libraries/dos.h>
#include "cycles.h"
#ifdef AZTEC_C
extern int      Enable_Abort;
#endif
extern BYTE     scr_image[];

int xdir[]      = { 0, 1, 0, -1 };      /* up, right, down, left */
int ydir[]      = { -1, 0, 1, 0 };      /* up, right, down, left */
int TIME        = 6;
int PLAYERS     = 2;
int GAMES       = 0;
int scores[MAX_COLOURS];

IMAGE   scr_data;               /* screen image data                    */
LIST    cycle_list;             /* list of current cycles               */
TIMEREQ *tr;                    /* timer ExtIO request structure        */
PORT    *sPort;                 /* server port                          */
PORT    *tPort;                 /* timer port                           */
int     num_players  = 0;       /* how many players are there           */
int     num_waiting  = 0;       /* how many are waiting for new game    */
int     num_alive    = 0;       /* number of cycles that are still moving */

extern struct Window *war_window;  /* our window c/w close gadget */

main(argc,argv)
int argc; char **argv;
{
    ORDER   *msg;               /* order from client                    */
    CYCLE   *cycle;             /* current cycle structure              */
    int portsig, closesig;      /* signal bits for our ports            */
    int signals;                /* received signals                     */
    int used_colours = 0;       /* bitmask showing which colours are used */
    int i;                      /* misc                                 */
    int games_played;

#ifdef AZTEC_C
    Enable_Abort = 0;
#endif
    /* initialize screen data structures, will tell clients the shape
     * of the screen by providing a pointer to this data
     */
    scr_data.x      = X_SIZE;
    scr_data.y      = Y_SIZE;
    scr_data.buf    = scr_image;
```

```
    tr              = 0;
    games_played    = 0;

    for (i = 1; i < argc; i++) {
        switch (argv[i][0]) {
            case 't':   TIME = atoi(argv[i] + 1);
                        break;
            case 'p':   PLAYERS = atoi(argv[i] + 1);
                        if(PLAYERS < 2) PLAYERS = 2;
                        break;
            case 'g':   GAMES = atoi(argv[i] + 1);
                        break;
        }
    }
    sPort = CreatePort("Cycle Server", 0L);
    if (!sPort) {
        printf("Error, can't open server port\n");
        terminate();
    }
    tr = Timer_Open(UNIT_MICROHZ, sPort);

    if (!tr) terminate();

    /* open up custom screen etc. . . */
    if (g_init()) terminate();

    draw_scores();

    /* do setup */
    g_restart();

    portsig   = (1 << (sPort->mp_SigBit));
    closesig  = (1 << war_window->UserPort->mp_SigBit);

    srand(time(0));  /* seed the random number generator */
    NewList(&cycle_list);
    Timer_Post(tr, TIME / 1000, (TIME % 1000) * 1000);

    for (;;) {
        /* wait for either a time event or a client */
        signals = Wait(closesig | portsig | SIGBREAKF_CTRL_C);

        if (signals & portsig) {
            while (msg = (ORDER *)GetMsg(sPort)) {

                if ((TIMEREQ *)msg == tr) { /* time request reply */
                    if (num_alive) move_cycles(&cycle_list);
                        Timer_Post(tr,
                                TIME * num_alive / 1000,            /* seconds  */
                                ((num_alive * TIME) % 1000) * 1000  /* micros   */
                                );
                    continue;
                }
                cycle = msg->cycle;

                if (!cycle) {
                    for (i = 0; i < 8; i++)
                        if ( (~used_colours) & (1 << i) ) break;

                    if (i == MAX_COLOURS) {
                        printf("Error, no colours left for the new cycle\n");
                        msg->status = STATUS_REMOVED;
                        ReplyMsg(msg);
                        continue;
                    }
                    cycle = AllocMem(sizeof(CYCLE), MEMF_PUBLIC);
                    if (!cycle) {
                        printf("Error, can't get memory for new cycle\n");
                        msg->status = STATUS_REMOVED;
                        ReplyMsg(msg);
                        continue;
                    }
                    AddHead(&cycle_list, &cycle->node);
                    num_players++;
                    msg->cycle      = cycle;
                    msg->table      = &scr_data;
                    cycle->status   = STATUS_LOSER;
                    cycle->dir      = DIR_RIGHT;
                    cycle->x        = rand() % (X_SIZE - 10) + 5;
                    cycle->y        = rand() % (Y_SIZE - 10) + 5;
                    cycle->colour   = i;
                    cycle->msg      = NULL;
                    used_colours    | = (1 << i);
                    scores[i]       = 0;
                }
                switch (msg->order) {
                    case ORD_DIRECTION:
                        if (cycle->status < STATUS_DEAD ) {
                            cycle->dir    = msg->dir;
                            cycle->status = STATUS_GOT_MOVE;
                            cycle->msg    = msg;
                        }
                        else {
                            msg->status  = cycle->status;
                            ReplyMsg(msg);
                        }
                        continue;
                    case ORD_AWAIT:
                        if (cycle->status < STATUS_DEAD)  /* still alive */
                            num_alive--;
                        cycle->msg    = msg;
                        cycle->status  = STATUS_AWAITING;
                        num_waiting++;
                        if (num_waiting == num_players) {
                            if (GAMES > 0 && games_played >= GAMES)
                                terminate();
                            else if (num_players >= PLAYERS) {
                                g_restart();
                                restart_cycles(&cycle_list);
                                games_played++;
                            }
                        }
                        continue;

                    case ORD_RETIRE:
                        Remove(cycle);
                        if (cycle->status < STATUS_DEAD)  /* still alive */
                            num_alive--;
                        FreeMem(cycle,sizeof(CYCLE));
                        num_players--;
                        used_colours   &= ~(1 << cycle->colour);
                        msg->status     = STATUS_REMOVED;
                        ReplyMsg(msg);
                        continue;

                    default:
                        printf("Got a invalid request.  Ignoring\n");
                        ReplyMsg(msg);
                        continue;
                }
            }
        }
        if (signals & SIGBREAKF_CTRL_C || signals & closesig)
            terminate();
    }
}

terminate()
{
    ORDER *msg;

    free_cycles(&cycle_list);

    if (tr) {
        WaitIO(tr);
        tr->tr_node.io_Message.mn_ReplyPort = 0L;
        Timer_Close(tr);
    }

    /* inform all players that are still active that we are shutting down */
    while (num_players) {
        WaitPort(sPort);
        while (msg = (ORDER *)GetMsg(sPort)) {
            msg->status = STATUS_REMOVED;
            /* if a new player tries do join, don't          */
            /* decrement the count the actual structure      */
            /* has already been free so don't look at it     */
            if (msg->cycle) num_players--;
            ReplyMsg(msg);
        }
    }
    g_finish();

    /* here we clear out the Port before deleting it, it is important to
     * Forbid() first so that no new messages arrive at the port between
     * the time we last look at it and the time it is DeletePort'ed
     */

    Forbid();
    while (msg = (ORDER *)GetMsg(sPort)) {
        msg->status = STATUS_REMOVED;
        ReplyMsg(msg);
    }
    DeletePort(sPort);
    Permit();
    exit(0);
}
```

Listing 1b: "dobikes.c"

```
/*
 * this code is for moving the cycles around on the screen and
 * updating their status at intervals
 */

#include "cycles.h"

extern int xdir[], ydir[];
extern int num_players;
extern int num_waiting;
extern int num_alive;

int ties;

move_cycles(list)
LIST *list;
{
    register CYCLE *c, *c2;
    int na;

    na  = num_alive;
    c   = (CYCLE *)list->lh_Head;
```

```
    if (c->node.ln_Succ) {      /* rotate the list so that     */
        Remove(c);              /* each cycle gets a turn      */
        AddTail(list,c);        /* at first reply              */
    }
    c = (CYCLE *)list->lh_Head;

    while (c2 = (CYCLE *)c->node.ln_Succ) {
        switch (c->status) {

        case STATUS_GOT_MOVE:
        case STATUS_NEED_NEW:
            link_pix(c->colour, c->x, c->y, c->dir);
            c->x  += xdir[c->dir];
            c->y  += ydir[c->dir];
            if (rpix(c->x,c->y)) {
                c->status = STATUS_LOSER;
                num_alive--;
                if (num_alive == 0) {
                    ties++;
                    draw_scores();
                    flash_colour(0);
                }
            }
            else {
                if (na>1)
                    c->status = STATUS_NEED_NEW;
                else {
                    scores[c->colour]++;
                    draw_scores();
                    c->status = STATUS_WINNER;
                    flash_colour(c->colour+3);
                    /* avoid title colours */
                    num_alive--;
                }
                wpix(c->colour, c->x, c->y, c->dir);
            }

            if (c->msg) {
                c->msg->x      = c->x;
                c->msg->y      = c->y;
                c->msg->dir    = c->dir;
                c->msg->status = c->status;
                ReplyMsg(c->msg);
                c->msg         = NULL;
            }
            break;

        default:
            break;
        }
        c = c2;
    }
}

free_cycles(list)
LIST *list;
{
    register CYCLE *c,*c2;

    c = (CYCLE *)list->lh_Head;

    /* check all cycles. . . */
    while (c2 = (CYCLE *)c->node.ln_Succ) {
        /* if there is any sort of message pending */
        if (c->msg) {
            /* inform the client that he has been terminated */
            c->msg->status = STATUS_REMOVED;
            ReplyMsg(c->msg);
            c->msg = NULL;
            num_players--;
        }

        /* clean up client. . . */
        Remove(c);
        FreeMem(c,sizeof(CYCLE));
        c = c2;
    }
}

restart_cycles(list)
LIST *list;
{
    register CYCLE *c,*c2;

    c = (CYCLE *)list->lh_Head;

    while (c2 = (CYCLE *)c->node.ln_Succ) {
        if (c->status == STATUS_AWAITING) {
            c->x           = rand() % (X_SIZE - 10) + 5;
            c->y           = rand() % (Y_SIZE - 10) + 5;
            c->dir         = DIR_RIGHT;
            c->status      = STATUS_NEED_NEW;
            c->msg->x      = c->x;
            c->msg->y      = c->y;
            c->msg->status = c->status;
            c->msg->dir    = c->dir;
            wpix(c->colour, c->x, c->y, c->dir);

            ReplyMsg(c->msg);
            c->msg = NULL;
        }
        c = c2;
    }
    num_waiting = 0;
    num_alive   = num_players;
}
```

Listing 1c: "graphic.c"

```
/*
 * Here we have all of the graphics rendering code. . .
 */

#include <exec/types.h>
#include <intuition/intuition.h>
#include "cycles.h"

#define INTUITION_REV    29
#define GRAPHICS_REV     29
#define X_BYTES ((X_SIZE/8)+!!(X_SIZE%8))

struct NewScreen newscreen = {
    0, 0,               /* left edge, top edge      */
    320, 200,           /* width, height            */
    3,                  /* depth                    */
    0,1,                /* DetailPen, BlockPen      */
    0,                  /* viewmodes                */
    CUSTOMSCREEN,       /* type                     */
    NULL,               /* font                     */
    (UBYTE *)" Cycles",  /* title                    */
    NULL,               /* Gadgets                  */
    NULL                /* CustomBitMap             */
};

struct NewWindow newwindow = {
    0, 0,               /* left edge, top edge          */
    15, 10,             /* width, height                */
    0, 1,               /* detail pen, block pen        */
    CLOSEWINDOW,        /* IDCMP Flags                  */
    WINDOWCLOSE,        /* flags                        */
    NULL,               /* first gadget                 */
    NULL,               /* checkmark                    */
    NULL,               /* title                        */
    NULL,               /* screen will be filled in later */
    NULL,               /* bitmap                       */
    0, 0,               /* min width, height            */
    -1, -1,             /* max width, height            */
    CUSTOMSCREEN        /* type                         */
};

UWORD scr_colours[] = {
    0x000, 0xf80,  0x002, 0xf00,
    0x00f, 0xfff,  0x0f0, 0xff0
};

BYTE scr_image[ X_BYTES * Y_SIZE];

struct IntuitionBase    *IntuitionBase;
struct GfxBase          *GfxBase;
struct Screen           *war_screen;
struct Window           *war_window;
struct RastPort         *rp;
struct Screen           *OpenScreen();
struct Window           *OpenWindow();
void                    *OpenLibrary();

g_init()
{
    IntuitionBase = OpenLibrary("intuition.library", INTUITION_REV);
    if (!IntuitionBase) {
        g_finish();
        printf("unable to open intuition\n");
        return(1);
    }
    GfxBase = OpenLibrary("graphics.library", GRAPHICS_REV);
    if (!GfxBase) {
        g_finish();
        printf("unable to open graphics library\n");
        return(1);
    }
    war_screen = OpenScreen(&newscreen);
    if (!war_screen) {
        g_finish();
        printf("unable to open screen\n");
        return(1);
    }
    LoadRGB4(&war_screen->ViewPort, scr_colours, 8);
    rp = &war_screen->RastPort;
    newwindow.Screen = war_screen;

    war_window = OpenWindow(&newwindow);
    if (!war_window) {
        g_finish();
        printf("unable to open window\n");
        return(1);
    }
    return(0);
}

g_restart()
```

```
{
    int i,j;
    for (i = 0; i < X_BYTES * Y_SIZE; i++) scr_image[i] = 0;

    SetAPen(rp, 0);
    RectFill(rp, 0, 10, 319, 199);
    SetAPen(rp, 1);
}

g_finish()
{
    if (war_window)     CloseWindow(war_window);
    if (war_screen)     CloseScreen(war_screen);
    if (GfxBase)        CloseLibrary(GfxBase);
    if (IntuitionBase)  CloseLibrary(IntuitionBase);
}

static int power[] = {1,2,4,8,16,32,64,128};

rpix(x,y)
{
    if (x < 0 || y < 0 || x >= X_SIZE || y >= Y_SIZE) return(1);
    return(scr_image[(x >> 3) + y * X_BYTES] & power[x & 7]);
}

typedef struct {
    int x,y;
} PAIR;

PAIR old_data[]    = { {4,7},{0,4},{4,0},{7,4} };
PAIR new_data[]    = { {4,0},{7,4},{4,7},{0,4} };

void link_pix(c, x, y, dir)
{
    int sx, sy;

    if (x < 0 || y < 0 || x >= X_SIZE || y >= Y_SIZE) return;

    SetAPen(rp, c + 3);   /* don't use title bar colours */
    sx  = x << 3;
    sy  = (y << 3) + 10;
    Move(rp, sx + 4, sy + 4);
    Draw(rp, sx + new_data[dir].x, sy + new_data[dir].y);
}

void wpix(c, x, y, dir)
{
    int sx, sy;

    if (x < 0 || y < 0 || x >= X_SIZE || y >= Y_SIZE) return;
    scr_image[(x >> 3) + y * X_BYTES] |= power[x & 7];
    SetAPen(rp, c + 3);  /* don't use title bar colours */
    sx  = x << 3;
    sy  = (y << 3) + 10;
    Move(rp, sx + old_data[dir].x, sy + old_data[dir].y);
    Draw(rp, sx + 4, sy + 4);
}

flash_colour(c)
{
    UWORD rgb;
    int i, red, grn, blu;
    struct ViewPort *vp;

    vp   = &war_screen->ViewPort;
    rgb  = GetRGB4(vp->ColorMap, c);
    red  = (rgb >> 8) & 0x0f;
    grn  = (rgb >> 4) & 0x0f;
    blu  = rgb        & 0x0f;

    for (i = 0; i < 10; i++) {
        Delay(5);
        WaitTOF();
        SetRGB4(vp, c, 8^red, 8^grn, 8^blu);
        Delay(5);
        WaitTOF();
        SetRGB4(vp, c, red, grn, blu);
    }
}

draw_scores()
{
    int i;
    char buf[20];
    extern int ties;

    SetBPen(rp, 1);
    SetDrMd(rp,JAM2);
    Move(rp, 90, 7);
    SetAPen(rp, 0);
    sprintf(buf,"%-3d ",ties);
    Text(rp, buf, strlen(buf));
    for (i = 0; i < MAX_COLOURS; i++) {
        SetAPen(rp, i + 3);
        sprintf(buf, "%-3d ", scores[i]);
        Text(rp, buf, strlen(buf));
    }
}
```

Listing 1d: "time.c"

```
/* "time.c"
 * Functions relating to the timer device: Timer_Open(), Timer_Close(), and
 * Timer_Post(). Timer_Post() requests the timer device to send a message after
 * the given number of seconds and microseconds.
 */
#include <exec/types.h>
#include <exec/ports.h>
#include <devices/timer.h>

#include "cycles.h"

TIMEREQ *Timer_Open(unit,tp)
ULONG unit;
PORT *tp;
{
    int error;
    TIMEREQ *tr;

    if (!tp) tp = CreatePort(0, 0);
    if (!tp) return(0);

    tr = (TIMEREQ *)CreateExtIO(tp, sizeof(TIMEREQ));
    if (!tr) {
        printf("Can't create timer request\n");
        return(0);
    }
    error = OpenDevice("timer.device", unit, tr, 0);
    if (error) {
        Timer_Close(tr);
        return(0);
    }
    return(tr);
}

/* close the timer device down */
void Timer_Close(tr)
TIMEREQ *tr;
{
    PORT *tp;

    if (tr) {
        tp = tr->tr_node.io_Message.mn_ReplyPort;
        if (tp) DeletePort(tp);
        CloseDevice(tr);
        DeleteExtIO(tr, sizeof(TIMEREQ));
    }
}

/* ask the timer to reply to our message after a certain time */
void Timer_Post(tr,secs,micro)
TIMEREQ *tr;
{
    tr->tr_node.io_Command   = TR_ADDREQUEST;
    tr->tr_time.tv_secs      = secs;
    tr->tr_time.tv_micro     = micro;
    SendIO(tr);
}
```

Listing 1e: "cycles.h" – the header file used by all of the server and manager programs.

```
/*
 * here we define all the structures and constants used in the server
 */
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/nodes.h>
#include <exec/ports.h>
#include <devices/timer.h>
#include <libraries/dos.h>

/* first some abbreviations that I like */
typedef struct Node          NODE;
typedef struct MsgPort       PORT;
typedef struct Message       MSG;
typedef struct List          LIST;
typedef struct timerequest   TIMEREQ;

/* info to describe each cycle */
typedef struct {
    NODE    node;          /* to keep things organized     */
    struct  Order *msg;    /* place for pending message    */
    int stat  us;          /* current status               */
    int x,y;               /* current position             */
    int dir;               /* current direction            */
    int colo  ur;          /* your colour designation      */
} CYCLE;

typedef struct {
    int  x;          /* x size of screen image   */
    int  y;          /* y size of screen image   */
    BYTE *buf;       /* pointer to buffer area   */
} IMAGE;

/* specification for the messages we expect to receive */
typedef struct Order {
    MSG   msg;     /* Exec message structure        */
    int   order;   /* command to be performed       */
    int   x,y;     /* parameters                    */
    int   dir;     /* direction                     */
    int   status;  /* your current status           */
```

```
    CYCLE *cycle;   /* pointer to my cycle structure   */
    IMAGE *table;   /* pointer to screen table         */
} ORDER;

#define STATUS_NEED_NEW      0 /* no direction change specified yet          */
#define STATUS_GOT_MOVE      1 /* direction change specified: will use       */

#define STATUS_DEAD        100 /* dummy entry -- bigger entries mean dead    */
#define STATUS_LOSER       101 /* you've crashed -- out of this game         */
#define STATUS_WINNER      102 /* you're the only remaining bike             */
#define STATUS_AWAITING    103 /* waiting for a new game                     */
#define STATUS_REMOVED     104 /* you are no longer in the game              */

#define ORD_DIRECTION        0 /* cycle change direction next move           */
#define ORD_AWAIT            1 /* reply when the next game has begun         */
#define ORD_RETIRE           2 /* I'm leaving --- forget all about me        */

#define TURN_LEFT(dir)       (((dir) - 1) & 3)
#define TURN_RIGHT(dir)      (((dir) + 1) & 3)

#define DIR_UP               0 /* constants for directions                   */
#define DIR_RIGHT            1 /* these are guaranteed never to change       */
#define DIR_DOWN             2 /* in future revisions so you can do          */
#define DIR_LEFT             3 /* things like adding 1 (mod 4) to turn       */
                               /* right, -1 to turn left. . .                */

#define MAX_COLOURS          5 /* maximum number of colours & players        */
#define X_SIZE              40 /* clients are not allowed to look at         */
#define Y_SIZE              23 /* these as the server may be compiled        */
                               /* with different values. The server          */
                               /* decides the size of the arena              */

extern MSG      *GetMsg();
extern PORT     *CreatePort();
extern void     *AllocMem();
extern void     *CreateExtIO();
extern TIMEREQ  *Timer_Open();
extern void     Timer_Close();
extern void     Timer_Post();

extern int  scores[];
```

Listing 2: This is the "manager" program that your player program should link with. The manager takes care of registering a player, putting up a window with his score and colour displayed, and closing everything down when the player exits. When using the manager with your player, you need only write the function strategy_init, strategy, and strategy_finish; see text for details.

```
/* "manager.c" */

#include "cycles.h"
#include <intuition/intuition.h>

#define PER_ROW    3 /* windows per row      */
#define HEIGHT    50 /* height of each window */

#ifdef AZTEC_C
extern int    Enable_Abort;
#endif
extern char  player_name[];

static struct Window   *war_window;
static struct RastPort *rp;

struct IntuitionBase   *IntuitionBase;
struct GfxBase         *GfxBase;

struct Window          *OpenWindow();
void                   *OpenLibrary();

int wins, losses;
int colour;

char *colours[] = { "Red", "Blue", "White", "Green", "Yellow" };

main()
{
    int status;
#ifdef AZTEC_C
    Enable_Abort = 0;
#endif
    if (Register(&colour)) exit(0);

    if (strategy_init()) {
        Retire();
        exit(0);
    }
    if (g_init((colour%PER_ROW)*(640/PER_ROW),(colour/PER_ROW)*HEIGHT)) {
        strategy_finish();
        Retire();
        exit(0);
    }
    g_show();
    status = STATUS_NEED_NEW;

    for (;;) {
        if (GetMsg(war_window->UserPort)) {
            strategy_finish();
            Retire();
            g_finish();
            exit(0);
        }
        switch (status) {
            case STATUS_WINNER:
                wins++;
                g_show();
                status = Await();
                break;
            case STATUS_LOSER:
                losses++;
                g_show();
                status = Await();
                break;
            case STATUS_NEED_NEW:
                status = strategy();
                if (status < 0) {
                    strategy_finish();
                    Retire();
                    g_finish();
                    exit(0);
                }
                status = Direction(status);
                break;
            case STATUS_REMOVED:
                WaitPort(war_window->UserPort);
                GetMsg(war_window->UserPort);
                strategy_finish();
                Retire();
                g_finish();
                exit(0);
        }
    }
}

/*
 * here we have all of the graphics rendering code for the player. . .
 */

static struct NewWindow newwindow = {
    0, 0,                          /* left edge, top edge          */
    640 / PER_ROW - 2, HEIGHT - 1, /* width, height                */
    0, 1,                          /* detail pen, block pen        */
    CLOSEWINDOW,                   /* IDCMP Flags                  */
    WINDOWCLOSE                    /* flags                        */
    | NOCAREREFRESH
    | SMART_REFRESH
    | WINDOWDRAG
    | WINDOWDEPTH,
    NULL,                          /* first gadget                 */
    NULL,                          /* checkmark                    */
    (UBYTE *)player_name,          /* title                        */
    NULL,                          /* screen will be filled in later */
    NULL,                          /* bitmap                       */
    0, 0,                          /* min width, height            */
    -1, -1,                        /* max width, height            */
    WBENCHSCREEN                   /* type                         */
};

g_init(x,y)
{
    IntuitionBase = OpenLibrary("intuition.library", 0);
    if (!IntuitionBase) {
        g_finish();
        printf("unable to open intuition\n");
        return(1);
    }
    GfxBase = OpenLibrary("graphics.library", 0);
    if (!GfxBase) {
        g_finish();
        printf("unable to open graphics library\n");
        return(1);
    }
    newwindow.LeftEdge  = x;
    newwindow.TopEdge   = y + 20;
    war_window = OpenWindow(&newwindow);
    if (!war_window) {
        g_finish();
        printf("unable to open window\n");
        return(1);
    }
    rp = war_window->RPort;
    SetBPen(rp, 0);
    SetAPen(rp, 1);
    SetDrMd(rp, JAM2);
    return(0);
}

g_finish()
{
    if (war_window)  CloseWindow(war_window);
    if (GfxBase)     CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);
}

g_show()
{
    char buf[40];
    int y  = 8;
```

```
    int x   = 8;

    Move(rp, x, y += 9);
    sprintf(buf, "%s player", colours[colour]);
    Text(rp,buf, strlen(buf));

    Move(rp, x, y += 9);
    sprintf(buf, "   Wins: %d", wins);
    Text(rp, buf, strlen(buf));

    Move(rp, x, y += 9);
    sprintf(buf, "Losses: %d", losses);
    Text(rp, buf, strlen(buf));

    Move(rp, x, y += 9);
    sprintf(buf, " Games: %d", wins + losses);
    Text(rp, buf, strlen(buf));
}
```

Listing 3: "play.c" – routines provided for the convenience of player programs.

```
/*
   This module contains the functions that players can use to control their
   movement and find out about the playing arena. These are:

   Register()        used to register in the game (done by the manager)
   Retire()          exit the game (also done by the manager)
   Inquire()         find out a given cell in the arena is obstructed
   GetScreenMem()    return pointer to the arena bitmap
   Direction()       specify new direction
   Await()           wait until new game begins
   GetInfo()         give player's current position and direction
   Look()            look in given direction and return status of cell there
*/
#include "cycles.h"
#ifdef AZTEC_C
extern int Enable_Abort;
#endif
PORT *FindPort();

static ORDER  order;
static PORT   *sPort;
static PORT   *cPort;
static BYTE   *screen;
static int    x_size, y_size, x_bytes;
static int    colour;

static xdir[]  = {0,1,0,-1};   /* up, right, down, left */
static ydir[]  = {-1,0,1,0};   /* up, right, down, left */

/* send a message to the target and wait for a reply */
MSG *Send(msg)
MSG *msg;
{
    PutMsg(sPort,msg);
    WaitPort(cPort);
    return(GetMsg(cPort));
}

/* register this player with the cycle server returns          */
/* non-zero if registration is unsuccessful also tells         */
/* you which colour you will be playing (for output)           */

Register(my_colour)
int *my_colour;
{
    cPort = CreatePort(0,0);
    if (!cPort) {
        printf("Can't create a reply port\n");
        return(-1);
    }

    order.msg.mn_Node.ln_Type  = NT_MESSAGE;
    order.msg.mn_ReplyPort     = cPort;
    order.order                = ORD_AWAIT;
    order.cycle                = 0;

    /* it is critical that the port not disappear between the time
     * it is found and the first message is sent to it.
     * This will ensure that proper shutdown will always happen
     * if the server decides to terminate
     */

    Forbid();
    sPort = FindPort("Cycle Server");
    if (!sPort) {
        Permit();
        printf("Can't find Cycle Server Port\n");
        DeletePort(cPort);
        return(-1);
    }
    PutMsg(sPort,&order);
    Permit();

    WaitPort(cPort);
    GetMsg(cPort);

    if (order.status) {
        printf("Server shutting down.  No more clients accepted\n");
        DeletePort(cPort);
        return(-1);
    }

    colour      = order.cycle->colour;
    x_size      = order.table->x;
    y_size      = order.table->y;
    screen      = order.table->buf;
    x_bytes     = (x_size>>3)+ !!(x_size & 7);
    *my_colour  = colour;

    return(0);
}

/* retire from the game. . . free resources  */
/* used to participate; no diagnostics       */

void Retire()
{
    /* check to see if we're already gone. If so then no ORD_RETIRE */
    if (order.status != STATUS_REMOVED) {
        order.order = ORD_RETIRE;
        Send(&order);
    }
    DeletePort(cPort);
}

static int power[] = {1,2,4,8,16,32,64,128 };

/* return the status of the pixel @ x,y */
Inquire(x,y)
{
    if (x<0 || y<0 || x>=x_size || y>=y_size) return(1);
    return ((screen[(x>>3)+ y*x_bytes] & power[x&7]) != 0);
}

/* return the location of the screen. . . in case the client wishes
 * to do some custom testing
 */

IMAGE *GetScreenMem()
{
    return(order.table);
}

/* returns status from server, non-zero indicates              */
/* something exceptional e.g. LOSER or WINNER declared         */

Direction(dir)
{
    order.order  = ORD_DIRECTION;
    order.dir    = dir;
    Send(&order);
    return(order.status);
}

/* wait for new game to start */
Await()
{
    order.order = ORD_AWAIT;
    Send(&order);
    return(order.status);
}

/* get last known information about self from server */
GetInfo(x, y, dir)
int *x, *y, *dir;
{
    *x   = order.x;
    *y   = order.y;
    *dir = order.dir;
}

/* look in the given direction for a pixel */
Look(dir)
int dir;
{
    return (Inquire(order.x + xdir[dir], order.y + ydir[dir]));
}
```

Listing 4: This is the "makefile" for use with Manx's make utility. It will compile the server ("cycle") or any or all of the player programs.

```
CFLAGS= +L -Iram:symbols.m

all: ram:symbols.m ram:c32.lib cycle\
    player1 player2 player3 player4 player5 player6 player7 player8

OBJS= cycles.o time.o graphic.o dobikes.o

cycle:  $(OBJS)
    ln -o cycle $(OBJS) -lc32

ram:symbols.m:
    copy dh:include/symbols.m ram:symbols.m

ram:c32.lib:
    copy dh:lib/c32.lib ram:

player1: player1.o play.o manager.o
    ln -o player1 player1.o manager.o play.o -lc32

player2: player2.o play.o manager.o
    ln -o player2 player2.o manager.o play.o -lc32
```

```
player3: player3.o play.o manager.o
    ln -o player3 player3.o manager.o play.o -lc32

player4: player4.o play.o manager.o
    ln -o player4 player4.o manager.o play.o -lc32

player5: player5.o play.o manager.o
    ln -o player5 player5.o manager.o play.o -lc32

player6: player6.o play.o manager.o
    ln -o player6 player6.o manager.o play.o -lc32

player7: player7.o play.o manager.o
    ln -o player7 player7.o manager.o play.o -lc32

player8: player8.o play.o manager.o
    ln -o player8 player8.o manager.o play.o -lc32
```

The Players: Here are some sample player programs to give you an idea of how to write your own. Players can range from the ten-line simple-minded player to much smarter, more complex routines.

Player 1

```
/*
    An example of a simple player. As his name suggests, Mr. Right just tries
    to turn right if his way is blocked, and turns left if there's something
    in the way. As you can see, writing a simple player using the manager code
    is trivial.
*/
#include "cycles.h"

char player_name[] = "Mr. Right";

strategy_init() {}
strategy_finish() {}

strategy()
{
    int x,y,dir;

    GetInfo(&x,&y,&dir);
    if (!Look(dir))
        return(dir);

    if (!Look(TURN_RIGHT(dir)))
        return(TURN_RIGHT(dir));

    return(TURN_LEFT(dir));
}
```

Player 2

```
/*
    Another "featherweight" player. "Dizzy" tries to turn left; if that fails
    he tries to turn right; and if that fails he continues in his original
    direction.
*/
#include "cycles.h"

char player_name[] = "Dizzy";

strategy_init() {}
strategy_finish() {}

strategy()
{
    int x,y,dir;

    GetInfo(&x,&y,&dir);

    if (!Look(TURN_LEFT(dir)))
        return(TURN_LEFT(dir));
    else if (!Look(TURN_RIGHT(dir)))
        return(TURN_RIGHT(dir));
    else
        return(dir);
}
```

Player 3

```
/*
    Another simple player. "Vertigo" will try to turn if she is moving
    horizontally, but prefers to continue with her current heading if moving
    vertically. In other words, Vertigo prefers the vertical directions.
*/
#include "cycles.h"

char player_name[] = "Vertigo";

strategy_init() {}
strategy_finish() {}

strategy()
{
    int x,y,dir;

    GetInfo(&x,&y,&dir);

    /* (dir & 1) is true for left and right */
    if (dir & 1) {
        if (!Look(TURN_RIGHT(dir)))
            return(TURN_RIGHT(dir));
        else if (!Look(TURN_LEFT(dir)))
            return(TURN_LEFT(dir));
        else
            return(dir);
    }
    else {
        if (!Look(dir))
            return(dir);
        else if (!Look(TURN_RIGHT(dir)))
            return(TURN_RIGHT(dir));
        else
            return(TURN_LEFT(dir));
    }
}
```

Player 4

```
/*
    "Round the clock" is a lightweight – he looks at all the cells around his
    current position up to four cells away. A sum is computed for the cells in
    each direction, with the closer cells receiving a greater weight. The
    direction with the lowest "score" is then chosen. Such a strategy can be
    effective, but you must resist looking too many layers away and spending
    too long computing the next move.
*/
#include "cycles.h"

char player_name[] = "Round the Clock";

int xdir[] = { 0, 1, 0, -1 };
int ydir[] = { -1, 0, 1, 0 };

strategy_init() {}
strategy_finish() {}

strategy()
{
    int x, y, dir;
    int score, best;
    int i, j;

    GetInfo(&x, &y, &dir);

    best = 1000; /* bigger than biggest possible score */
    for (i = 0; i < 4; i++) {
        if (Look(i)) continue;
        score = 0;
        for (j = 2; j < 4; j++)
            score += Inquire(x + j*xdir[i], y + j*ydir[i]);
        if (score < best ) {
            best = score;
            dir = i;
        }
    }
    return(dir);
}
```

Player 5: "Keep 'em Flying" uses a similar strategy to "Round the Clock", above, but looks around starting from her current direction. This results in "Keep 'em Flying" preferring her current direction to a new one.

Player 6

```
/*
    "Not so Dizzy" is a middleweight player who evaluates the scores around
    him in the manner of "Round the Clock" and "Keep'em Flying", but with a
    more sophisticated evaluation algorithm. Notice that, for speed, the arena
    memory is searched directly instead of on a cell-by-cell basis via the
    Inquire() function: a local copy of the board is made so that it may be
    searched and modified without affecting the actual arena. The
    strategy_init and strategy_finish functions allocate and free the
    required memory.
*/
#include "cycles.h"

char player_name[] = "Not so Dizzy";

int xdir[] = { 0, 1, 0, -1 };
int ydir[] = { -1, 0, 1, 0 };

static IMAGE myTable;
static nbytes;
static IMAGE *table = 0;
static x_bytes;

void mySet(), myClr(), *malloc();
IMAGE *GetScreenMem();

strategy_init()
{
    table       = GetScreenMem();
    myTable.x   = table->x;
    myTable.y   = table->y;
    x_bytes     = (myTable.x>>3) + (!!(myTable.x&7));
    nbytes      = myTable.y * x_bytes;
    myTable.buf = (BYTE *)malloc(nbytes);
    if (!myTable.buf) {
        printf("Error, can't get memory for my copy of the board\n");
        return(-1);
    }
    return(0);
```

```
}

strategy_finish()
{
    free(myTable.buf);
}

strategy()
{
    int x, y, dir;
    int score, best;
    int i, j, d;

    GetInfo(&x, &y, &dir);
    best = 0;
    copyBoard();
    for (d = TURN_RIGHT(dir), i = 0; i < 4; i++, d=TURN_RIGHT(d)) {
        score = value(x + xdir[d], y + ydir[d], 2);
        if (score > best) {
            dir  = d;
            best = score;
        }
    }
    return(dir);
}

value(x, y, d)
{
    int i, best, v;

    if (myInquire(x, y)) return(0);
    if (d > 0) {
        mySet(x, y);
        best = 0;
        for (i = 0; i < 4; i++) {
            v = value(x + xdir[i], y + ydir[i], d-1);
            if (v > best) best = v;
        }
        myClr(x, y);
        return(1 + best);
    }
    return(1);
}

copyBoard()
{
    register i;
    register char *p, *q;

    q = myTable.buf;
    p = table->buf;
    for (i = nbytes; --i >= 0; )
        *q++ = *p++;
}

static power[] = {1, 2, 4, 8, 16, 32, 64, 128};

/* return the status of the pixel @ x,y */
myInquire (x, y)
{
    if (x < 0 || y < 0 || x >= myTable.x || y >= myTable.y) return(1);

    return ((myTable.buf[(x >> 3)+ y*x_bytes] & power[x & 7]) != 0);
}

/* return the status of the pixel @ x,y */
void mySet (x, y)
{
    if (x < 0 || y < 0 || x >= myTable.x || y >= myTable.y) return;
    myTable.buf[(x >> 3)+ y*x_bytes] |= power[x & 7];
}

/* return the status of the pixel @ x,y */
void myClr (x, y)
{
    if (x < 0 || y < 0 || x >= myTable.x || y >= myTable.y) return;
    myTable.buf[(x >> 3)+ y*x_bytes] &= ~power[x & 7];
}
```

Player 7: "Look B4 U Leap" uses a similar strategy to "Not So Dizzy" except that he prefers the current direction, and looks one cell further from his current position, looking ahead a total of four cells. This player consistently wins in games played in the 20 millisecond class.

Player 8

```
/*
    This player has no strategy of its own, but just moves in the direction
    indicated by a joystick; this lets a human play against the Warrior Cycle
    programs.
*/
#include "cycles.h"
#include <exec/types.h>
#include <exec/devices.h>
#include <devices/gameport.h>
#include <devices/inputevent.h>

char player_name[] = "Joystick";

static struct MsgPort      *gameport;
static struct IOStdReq     *gameio;
static BYTE                gamebuffer[sizeof(struct InputEvent)];
static BYTE                *gamebuff;
static struct InputEvent   *gamedata;

struct IOStdReq *CreateStdIO();

strategy_init()
{
    gameport = CreatePort(0,0);
    if (!gameport) {
        printf("Error, couldn't create port\n");
        return(-1);
    }
    gameio = CreateStdIO(gameport);
    if (!gameio) {
        printf("Error, couldn't create StdIO message\n");
        DeletePort(gameport);
        return(-1);
    }
    if (OpenDevice("gameport.device",1,gameio,0)) {
        printf("Error, couldn't open gameport.device\n");
        DeleteStdIO(gameio);
        DeletePort(gameport);
        return(-1);
    }
    if (SetCType(GPCT_ABSJOYSTICK)) {
        printf("Error, can't set controller type\n");
        CloseDevice(gameio);
        DeleteStdIO(gameio);
        DeletePort(gameport);
        return(-1);
    }
    if (SetTrigger(GPTF_UPKEYS|GPTF_DOWNKEYS,50)) {
        printf("Error, can't set trigger type\n");
        CloseDevice(gameio);
        DeleteStdIO(gameio);
        DeletePort(gameport);
        return(-1);
    }
    gameio->io_Command  = GPD_READEVENT;
    gameio->io_Data     = (APTR)gamebuffer;
    gameio->io_Length   = sizeof(struct InputEvent);
    gameio->io_Flags    = 0;
    gamedata            = (struct InputEvent *)gamebuffer;
}

strategy_finish()
{
    CloseDevice(gameio);
    DeleteStdIO(gameio);
    DeletePort(gameport);
}

strategy()
{
    int code;
    int xmove, ymove;
    int x,y,dir;

    GetInfo(&x, &y, &dir);
    gameio->io_Length  = sizeof(struct InputEvent);
    SendIO(gameio);
    WaitPort(gameport);
    /* wait for joystick event, chain to last event in list */
    while (GetMsg(gameport))
        ;
    code    = gamedata->ie_Code;
    xmove   = gamedata->ie_X;
    ymove   = gamedata->ie_Y;
    if (ymove) return( (ymove<0) ? DIR_UP : DIR_DOWN );
    if (xmove) return( (xmove<0) ? DIR_LEFT : DIR_RIGHT );
    return(dir);
}

static SetCType(type)
{
    gameio->io_Command  = GPD_SETCTYPE;
    gameio->io_Length   = 1;
    gameio->io_Data     = (APTR)gamebuff;
    *gamebuff           = type;
    SendIO(gameio);
    WaitPort(gameport);
    GetMsg(gameport);
    return(gameio->io_Error);
}

static SetTrigger(keys,timeout)
{
    struct GamePortTrigger gpt;

    gameio->io_Command  = GPD_SETTRIGGER;
    gameio->io_Length   = sizeof(gpt);
    gameio->io_Data     = (APTR)&gpt;
    gpt.gpt_Keys        = keys;
    gpt.gpt_Timeout     = timeout;
    gpt.gpt_XDelta      = 1;
    gpt.gpt_YDelta      = 1;
    return(DoIO(gameio));
}
```

Reviews

Lattice C Compiler Version 4.00

Review by Chris Zamara

From: Lattice Incorporated
2500 S. Highland Avenue
Lombard, IL
60148 USA

Price: $200 (US) – C Compiler
$375 (US) – C Development System

When the first Lattice C compiler arrived with the Amiga, it had no competition and was blessed as the machine's official C development system. That didn't stop people from complaining about its slow compile times and large generated code sizes, however, and when Manx's Aztec C came out, Lattice looked even worse in comparison. Manx compiled faster, its linker was several times faster than Alink that came with Lattice, it produced better code, and was easier to use. Aztec soon became the system of choice for most serious developers.

With the new Lattice release of version 4.00, that may very well change. Since John Toebes (pronounced "TOE–bis") of Software Distillery fame joined Lattice, their Amiga C compiler has gone through major changes and enhancements to address the complaints against it. Whether it's now "better" than Manx depends on where your priorities lie, as you'll see from reading on, but it's obvious that every weak point that the compiler had has been considerably strengthened, and enough new features have been added to get Amiga C programmers excited.

In a nutshell, the chief improvements to the compiler over previous Lattice versions are: faster compiling, better optimized code, built-in functions, direct calling of system functions (without using the stub routines in Amiga.lib), improved error messages, support for short (16-bit) integers, ANSI-standard compliance (including function prototyping), compressed header files, and more library functions. It's also easier to invoke from the CLI, offering options to compile and link with a single command, or to only compile modules that have been changed since the previous compile. What's more, the linker included is an improved version of The Software Distillery's Blink, and it supports true overlays, pre-linking, library indexing for better link times, and interactive link-time symbol definition. The package comes with an improved assembler that supports the standard include files, and some public domain utilities of interest to programmers.

Version 4.00 of Lattice is an important product for all Amiga C programmers, so this review will not only give a thumbs up or down on the product, but also explain all the new features in turn to help you make up your mind.

System Routine Calls With "#pragma"

An innovative optimization is the optional way of handling calls to system functions. The code generated by C to call a system function has never been very efficient on the Amiga because system routines expect their parameters in registers, while C passes them on the stack. To get around this, you always linked with a library (amiga.lib) of short system routine interface functions that pulled parameters off the stack, put them into the appropriate registers, and called the real system function. This unproductive overhead for each system call encouraged assembler programmers to sneer at the C faithful; now users of Lattice 4.00 can sneer right back.

A new pre-processor directive called "#pragma" allows you to define a system function's offset in the library table and the register conventions for calling the function. Actually, you never have to make these definitions yourself because header files come on the system disks with #pragma statements for all system functions. When a #pragma exists for a function, the compiler generates code to call it directly, without the useless overhead of an extra function call, stack popping and register assignment. What's more, using #pragma statements for all the system functions your program is calling means you don't have to link with the "amiga.lib" library of hook-up functions, shortening linking time. This can be a major advantage if you don't have extra RAM or a hard drive, and have to link with the libraries on floppy. The fact that the #pragma statments are non-portable between compilers isn't much of a problem, since removing them from your source won't affect the program's operation, just its efficiency. Transporting your code to another compiler will just mean removing a single #include statement from your source.

To keep the #pragma statements current with new operating system revisions, a utility is included called "fd2pragma". As its name suggests, this program takes the ".fd" file from the "Extras" system disk as input and creates header files containing the appropriate #pragma commands. The .fd files give the offsets and register conventions for all system routines, and are used for calling libary routines from AmigaBasic. Using the .fd files to generate the #pragma headers is a good approach: since the fd-file format is known and everyone has access to them, they are an excellent source of information about the current state of system library routines.

Function Prototyping

Another aid to function-calling is Lattice's function prototyping, a feature that conforms to the proposed ANSI standard for C compilers. When declaring a function, you can also declare the types of all its parameters. The same header files that contain the #pragma statements have the prototypes for all system functions. Using the prototypes can really speed up program development: a great many program crashes can be traced to system routine calls that pass a short instead of a long integer, or pass the wrong number of parameters. By including the function prototypes, the proper conversions will be made between short or long integers, or you will get compile-time warnings of any incorrect function calls, including those for your own functions if you declare them with the prototype syntax. To be totally safe, you can use a compiler option to generate a warning for the call of any function that has no prototype declaration for it. Prototyping is an important feature for Lattice C, because it is a concrete benefit that Manx doesn't have.

Another level of ANSI-standard function checking is performed as well: unless a function specifically returns an INT, it must be declared. Functions that have no RETURN statement or have a return without a value must be declared and defined as VOID; if they're not, you get a warning message. Many programmers don't bother declaring functions that return nothing (especially main()!), and the warnings for every function can be maddening. Lattice thought of that, though, and the suppression of these warnings (and not other important ones) can be forced with one of the options that allows slight deviations from the ANSI standard. My first reaction to the warnings was negative, but when I saw how Lattice kept ANSI compatibility and at the same time kept programmers happy, I was impressed.

Built-ins

System routine calls are not the only ones to be streamlined; commonly used standard functions are now built-in, so that inline code can be generated to lessen the overhead of calling a general routine. Functions available as built-ins are those which, in the words of the manual, are "encouraged by the ANSI standard over others which might have previously been used": strlen, strcmp, strcpy, memset, memcmp, and memcpy. To access the built-in functions, the function name must be preceded with the string "_builtin_". This is accomplished by defines in a header file. Again, portability with other C implementations will not suffer, since your code itself doesn't have to change to use the built-ins. Since functions like strlen are used so often, it makes sense for the compiler to handle them directly to generate more efficient machine code. (For example, a call to strlen with a string constant as the parameter generates no code at all, but merely compiles to a constant.) If you want to call your own function instead of using the built-in or the library routine, it is still possible, so there is no disadvantage with the scheme.

Compressed Header Files

Compiling on the Amiga means compiling large header files along with your program. In fact, with a short program using, for example, Intuition's header files, more time can be spent compiling the header files than the program itself. One solution to the speed problem is to copy the header files to the RAM disk, but that takes a lot of room. A partial solution is offered by Lattice's compressed header file format: compressed header files are much smaller, and compile faster. All the system header files are available on the Lattice disk in compressed form, and a program called "lcompact" is supplied to compress standard header files. Compressed header files help, but are not as good a solution as Manx's pre-compiled header files. It may not seem fair to criticize Lattice because the competition has an extra feature, but since there are only two commercial C compilers for the Amiga, it is inevitable that one will be evaluated in terms of the other; after all, your choice comes down to one or the other. With that in mind, Lattice's lack of pre-compiled header files (or Manx's support of them) is a strong argument in favour of Manx. By pre-compiling the headers you need to use, you can compile your program without needing the header files around, and worse, waiting for them to compile every time.

Blink

One of the most vital improvements lies not with the compiler itself, but the linker that's bundled with it. The original Lattice came with Metacomco's horrendously slow Alink, which led The Software Distillery to come out with the much better and faster public-domain Blink. (In my tests, Blink linked faster than the Manx linker "ln".) Lattice has obtained the rights to distribute Blink, and it comes in the new C package with further improvements, fully documented in the single Lattice manual. Blink has many options for creating the executable program file, including the option to put DATA hunks in chip ram, coalesce all CODE or DATA hunks into a single hunk, handle overlays, suppress symbol table information, etc. A unique feature is its link-time definition of symbols – if Blink finds a symbol it can't resolve, it allows the user to enter a DEFINE value for the symbol instead of aborting. That way, if you spelled a function or variable name wrong in your source, you can test the program anyway without recompiling, then fix the spelling the next time you make changes. Not the most vital feature for a linker, perhaps, but it shows that Lattice is really trying hard to cater to a programmer's every desire.

With pre-linking, you can consolidate a number of object files into one that can be linked with the object modules that you're currently working on. This avoids linking many files every time when you just need a few.

The run-time libraries that come with the system are *indexed*, a format that Blink recognizes and uses to considerably speed up link times. You can, however, still link with the standard-format libraries, which also come with the system. This is an advantage over Manx's linker, which can *only* use libraries in its own format and requires the use of a conversion utility to make the standard "amiga.lib" library readable by its linker.

Setting up and Using the System

The system consists of four disks, but only two are needed to start compiling; the programs for compiling and linking are on the first disk, and the libraries and compressed header files are on the second. The easiest way to get started is to boot with disk one and put disk two in the second drive (Lattice recommends using a (minimum) two-drive system with the package). If you have a hard drive, you can use an execute script provided on the disk to put the required directories and files onto it and make the necessary ASSIGNs.

The system uses DOS virtual devices (using ASSIGN) to access the directories for the programs, header files, and libraries. This system works well, but it is not as flexible or unobtrusive as the environment variables – variables in a special library that can be set with a "set" command – used by Manx's Aztec C. For example, a search path for header files can be specified with the Manx INCLUDE variable, but only a single directory can be specified by assigning the virtual device INCLUDE:. Also, using the required assigns for INCLUDE, QUAD, LIB and LC obviously precludes using those names for something else. A small point, perhaps, but a difference worth noting.

A strong point of this Lattice release is how easy it is to compile and link a program. The "lc" command used to invoke both compiler passes will also invoke the linker if the "–L" option is specified. Also, any number of files can be compiled and linked with a single command. (The Manx compiler "cc" will only compile one file and has to be re-run for every module.) The entire compiling and linking process for a program composed of several modules can be brought about with the command:

```
lc -L module1 module2 . . .
```

Or, easier still, you can use standard AmigaDOS wildcard characters to match file names. For example, to compile every C source file in the current directory and link them with the standard libraries to create an executable program, you would use the command:

```
lc -L #?
```

What's more, the powerful "lc" front-end for the compiler system has an option that gives you some of the functionality of a "make" utility. The "–M" option forces the program to compile only those modules that are more recent than their corresponding object (.o) files. In other words, it will only compile the files that have been changed since the last compile, so you never compile more than is necessary. Although a true make utility (which you can also get from Lattice) is far more flexible, this option provides an easy way to develop programs consisting of many source files.

You can, of course, compile and then link with Blink separately, but for hassle-free program development, you can't beat "lc –L". If you do link separately, that task is made easier by the ".lnk" file created by the compiler; just "Blink WITH progname.lnk" and the linker gets all the right parameters regarding object files, libraries and options. The ".lnk" file is not removed after linking, so you can easily re-link after compiling just one module.

The lc command is actually a front-end to all the Lattice utilities, and will invoke the Object Module Librarian program to create your own library if that option is specified. The options you give to lc are passed on as the appropriate options for the programs that it runs.

One of lc's most important options lets you specify what kind of memory – chip or fast – the code, data, and uninitialized data segments are to go in. Being able to specify this via the compiler for selected modules is much better than Manx's approach of using a linker option, since the latter requires you to commit all modules together to the same type of RAM. (Data that is used by the Amiga's custom chips, such as graphics data, must be stored in chip RAM. Unless the programmer specifically ensures this, a program will not run properly on systems with expanded memory.)

The Package

The basic Lattice 4.0 compiler package consists of the manual and four disks and sells for $200.00 (all prices are given here in U.S. dollars). A full-blown "Lattice AmigaDOS C Development System" is available for $375 and includes, along with the compiler, text management utilities, a "make" utility, the Lattice Screen Editor, and Metadigm's excellent MetaScope debugger. Unless you already have similar utilities, the Development System looks like an excellent deal. This review, however, concentrates on the compiler system only.

Lattice's upgrade policy for registered users of older versions is quite reasonable: anyone who purchased a Lattice C compiler after August 1, 1987 gets a free upgrade to Version 4.0. Users of Version 3.1 can get the upgrade for $45.00, and all earlier versions can be upgraded for $75.00. You also have the option of upgrading the compiler alone to the whole Development System for $250.00.

The thick coil-bound manual is straightforward, starting with installation, proceeding to the system's special features, then going on to the good reference stuff that programmers live on. The library functions are divided into several classes: AMIGA, ANSI, LATTICE, UNIX, and XENIX. Functions in a given class are compatible with that standard; for example, any function in the class LATTICE will work the same way on any computer that Lattice makes a compiler for. The function descriptions are in alphabetical order, with an index, and function names in bold at the top of each page make it easy to find things by browsing. The only problem is that some functions are included with others and can't be found independently; to find *sprintf* or *printf*, for example, you have to know to look under *fprintf*, as they are not listed in the top-of-page area or in the index. This nitpick aside, the manual is top-notch in quality, accuracy and clarity.

Although the Lattice compiler, unlike Manx's, produces object code directly without going through an assembler, an assembler is provided for your own use. The new assembler is compatible with the standard syntax (as in the original MetaComCo assembler), works with the standard include files, and can be used with the supplied version of Blink. A section of the manual explains how to write modules in assembler and interface them with your C programs.

Some useful freely redistributable utilities – "ConMan", and "PopCLI", and "MemWatch" – come on the disk; ConMan and PopCLI are installed when you boot. You may have heard of these popular programs before or have them already. ConMan lets you edit previous commands in a CLI window or any console, PopCLI lets you pop up a new CLI window at any time by pressing a "hot key", and MemWatch monitors low memory to make sure a program isn't stepping anywhere it shouldn't be. The programs are available free from other sources, but getting them with the Lattice package may save you some downloading time.

Source code for Lattice's standard startup and exit code is also provided, as well as several programming examples and commented header files for C and assembler.

Technical support for the product is provided in three ways: on-line through BIX or the Lattice Bulletin Board Service, or by voice through the Technical Support Hotline. A $10 credit for BIX registration or time on an existing BIX account comes with the package. The Lattice BBS is open 24 hours, and although you can use any terminal program to log on, Lattice's Telecommunications package, *SideTalk* is pre-configured to use the service – a $40 coupon towards the purchase of SideTalk is included with the compiler. The tech support hotline (not a toll-free number) is open 9 AM to 4 PM, Monday through Friday. All the support information and coupons are right at the beginning of the manual so you can always find where to turn if all else fails. They recommend reading the manual before calling with technical questions, of course.

Compared with the Competition

Enough with the new features; how much better is the new version in terms of performance? According to Lattice, much better. They claim that the Dhrystone benchmark, which showed results of 460 with version 3.03, jumped to 730 with version 3.10, and is now up to 1295 with the new 4.0 – a 70 percent increase in speed. These are the results that Lattice claims for the standard Dhrystone, Float and Savage benchmarks compared with Manx version 3.40:

	Lattice 4.0	Manx 3.40
Dhrystone	1294 Dhrystones/sec	1010 Dhrystones/sec
Float	22.20 Sec (IEEE format) 10.16 Sec (FFP format)	98.85 Sec (IEEE format) 17.60 Sec (FFP format)
Savage	47.67 Sec/0.000000318 accuracy	119.6 Sec/0.000109 (IEEE) accuracy

Our test results with the Dhrystone program from Fish Disk #1 show different results than Lattice's Dhrystone figures above; Lattice still comes out ahead, but not by as big a margin. Using 16-bit integers with both compilers, the differences are even smaller. We're not sure, though, how meaningful any single benchmark is; the benchmark game is a complex one, and a handful of numbers doesn't begin to point out all the strengths and weaknesses of one compiler's generated code with respect to another's.

With that said, here are results from another benchmark. It's not a standard one, but it's designed to show code performance in specific kinds of common operations. Lack of space prevents us from listing the program here, but it will appear on the disk for this issue. It iterates three routines, one at a time, and records the time for all iterations of each routine. The routines are designed to test common operations in integer arithmetic, structures and pointers, and string handling. Here are the results of the test, using 10,000 iterations.

		Number of Ticks	
		Lattice	Aztec
Test:	Integer	243	344
	Pointer	352	407
	String	1045*	1011
	Total	1640	1762

(* using built-in string functions: 1032)

In this test at least, it seems that Lattice-generated code performs significantly better in integer math and slightly better in structure and pointer operations, but Manx has the upper hand in common string operations. Using the built-in string functions gives Lattice an improvement, but it still doesn't catch up with Manx's figure. This benchmark program is not exactly what you'd call scientific, but it agrees with the standard ones in that Lattice still comes out ahead overall.

Compile and Link Speed

The old Lattice was known for its slow compile times; in various side-by-side tests using the supplied compressed header files, we found version 4.0 to be

in the same ballpark as Manx 3.40. Results for compiling and linking a "typical" three-module program with a total of about a thousand lines of source are one minute 27 seconds with Aztec, and one minute 36 seconds with Lattice. (These tests were done with everything in the RAM disk to minimize the effect of different head-seek times for different files.) Generally, Lattice compiles slower than Manx, but Blink links faster and closes the gap. It really depends on what you're compiling, but they're generally close enough that compile time is no longer an issue.

What *is* an issue is whether or not you use Manx's pre-compiled header files when compiling – then the differences are more dramatic. It's not an unfair comparison, since you'll usually be using pre-compiled headers when you compile; many Manx users never bother with includes of system headers, and just compile with a pre-compiled file containing every system header they'll ever need. When you consider that a pre-compiled header may fit into RAM when all the header files won't (all Lattice's compressed headers are over 440k, while all headers in Manx pre-compiled format are about 118k), the difference in compile/link time is much greater.

One of the things that makes Lattice compile slower might also create other problems – the compiler itself is quite a bit larger. To compile, you need the front-end program "lc", and the two programs it runs called "lc1" and "lc2". The combined size of these programs is over 200k, while Manx's single "cc" program is less than 80k. This may be a factor for users with only 512k of RAM because you might be forced to load the programs from floppy for every compile, adding a noticeable amount of time to the process.

Looking at the size of the output code, Manx was the clear winner in smaller programs, often producing code up to 30 percent smaller. This has a lot to do with Lattice's larger overhead; a null program (main(){}) is 3892 bytes with Lattice, and only 1832 with Manx. This is because Lattice programs do more favours for you in their startup and exit code, like providing a requester for CTRL-C user aborts. This startup code can be changed or replaced with your own routines, so the extra overhead should not be held against Lattice. Even taking the overhead into account, though, the size of the executable program is often larger. The same 100-line program used in the above compile-speed test turned out to be 13,584 bytes long using Lattice and 9,852 with Manx.

For some programs, the difference isn't as great: compiling Transactor's "Structure Browser" version 1.1 with both compilers (using the 16-bit INT model) resulted in a 28,280 byte program from Manx and 30,456 from Lattice – only about eight percent larger. The smaller difference in this case was probably due to the fact that the program consists largely of static data rather than code. Overall, it is fair to say that Lattice produces cosistently larger code than Manx, even when using the options for SMALLCODE, SMALLDATA, PC-relative addressing, 16-bit ints and no stack-checking.

We hate to make sweeping generalizations like these, but from a variety of tests, here's what the comparison between Lattice 4.0 and Manx 3.6 looks like: Lattice code executes faster, Manx makes smaller code, and they're in the same ballpark with respect to compile/link time. These aren't the only important statistics of a compiler, however – there are some things to consider that aren't so cut and dried.

Error and Warning Messages

Lattice is much more finicky than Aztec, and will give you warnings over things in your code that you might not think twice about. In general, this is good, which is why the warnings are there in the first place. By compiling a large project with Lattice that was developed using Manx, we were able to catch some deeply-buried bugs that weren't found even in testing! Some of the warnings you'll get are uninitialized AUTO variables (a common source of hard-to-trace bugs), statements that have no effect (like coding c == 0 instead of c = 0), variables that are declared but never referenced (good for cleaning up old code), and others. Since C is generally so lax in enforcing rules, Lattice's vigilance can help keep you out of trouble. On the other hand, a common sentiment among dyed-in-the-wool C programmers is, "If I wanted LINT, I would have typed *lint*". Well, perhaps I'm not a macho C guy, but I'd rather catch potential errors at compile rather than run-time, thank you. I therefore look at Lattice's extra dose of warnings as a plus.

Incidentally, the error reporting in this version has been improved; the offending source code (or code nearby, anyway) is printed along with the error message to give you an idea of what happened before you dive back into the source. We still prefer Manx's page-at-a-time approach to displaying errors, but a CTRL-C will always kill a compile at any time (and even give you a chance at continuing before aborting if you wish).

A feature of Manx that some will miss more than others is the ability to put assembler code in-line with your C source with the #asm directive. You can still link with assembler modules, so Lattice doesn't really present a limitation in this area, just an inconvenience. How much of an inconvenience it is really depends on your personal programming habits.

Library Functions

The Lattice library has a very rich set of functions, covering I/O, DOS routines, string and integer operations, date and time functions, memory allocation, conversion between data types, floating point and trigonometric functions, etc. There are well over 100 functions, each concisely defined in the manual. Many functions are for Unix compatibility, and some do things that you could do just as easily with AmigaDOS functions, but they do make it easy to port Unix programs to the Amiga. Without going into a direct comparison, it is fair to say that the Lattice library is much more complete than Manx's, especially in string handling. When the next Manx release – Version 3.60 – arrives, which should be soon, we will do a thorough comparison of the libraries.

Your Best Bet?

The bottom line: should you get Lattice 4.0 or Manx 3.40? Or wait for Manx 3.60? Or wait for something else? Hopefully, you'll be able to assess your needs and make a choice based on what you now know. We can't leave you there, though – some sort of recommendation would be in order here. Let's look at your options.

If you don't currently have a C compiler and are trying to decide which one to go with, you're in an enviable but difficult position. Enviable because both commercial compiler packages available for the Amiga are so good; difficult because the decision is so hard to make. The price is about the same (looking at Manx's base system), so that won't help you decide. Neither one is better than the other in all areas, so you can't pick by high score.

It really comes down to the feel of each system, and it looks like for the new C programmer, or those who like a stricter language with less room for making errors, Lattice has the better feel. The compiler will take you by the hand and show you the error of your ways, probably leading to more thorough programming practices on your part. You will be able to produce bug-free code more easily, leading to faster development times in general. It's also easier to use, letting you compile multi-module programs using a single command, without having to learn the strange syntax of "make" (which the base version of Aztec C doesn't come with, by the way). The default startup and exit code will make your programs more well-behaved, even if it is larger than if compiled with Manx.

On the other hand, if you're an old C hand, perhaps with some leanings towards 68000 assembler, and want a lean, mean, compilin' machine that doesn't bother you with silly warnings and makes the smallest possible code, you may take the path leading to Manx. You won't have to worry about what header files you need for everything, and if you make dumb

little errors in your code, well, you'll find out when the program doesn't run right – at least you won't be bothered by ugly messages during compilation. You know what you're doing, and if you pass a short integer to a function that wants a long, you get what you deserve. You like to be able to optimize often-used functions simply by putting assembler code in-line, and being able to see the assembler file produced by the compiler is a good way to see the code that needs optimizing. You'd find Lattice too strict, too big, too removed from the action.

Perhaps a harder group to answer for are those currently using Manx – is it worth the switch? You would be losing your pre-compiled header files (which some people don't use anyway), and if you're used to running the compiler and linker from RAM, you may not have room for them any more. You would gain better error reporting, faster code, and more library functions. The programs you produce will often be larger.

It's easy enough to leave the decision to you; what about us – are we going to switch to Lattice 4.0? For some applications, probably, but certain realities will for the most part keep us with our trusty Aztec C. For one, we have lots of existing code that we don't feel like porting – many modules don't contain the necessary includes because they assume pre-compiled header files. There are other subtle differences, like Lattice's limit of 256 characters for string constants, that are enough to prevent existing code from compiling. Another factor that keeps us hooked on Manx is the promise of a source level debugger to be released with Version 3.60 – at least with that it'll be easy to *find* the bugs that Aztec makes it so easy to create. Finally, with the size of some of our projects, it would be hard to get along without "make". Lattice's "–M" option is nice, but doesn't automatically re-compile modules whose header files change, and we're not sure that it's worth the extra bucks for the developer's system just to get Lattice's Make utility.

So, although Lattice C 4.0 is a significant enough improvement to make a programmer's life easier, it sure makes his or her choices a lot tougher. That makes it all the more fortunate that, in this case, there's no such thing as a bad choice.

Power Windows 2.0

A code-generation utility for Intuition data structures

Review by Chris Zamara

From: Inovatronics
11311 Stemmons Frwy., Suite 8
Dallas, TX 75229

Price: $89.95 (US)

Once in a while, a utility comes along that improves your programming environment so much that it completely changes the way you look at and use your computer. You can recognize one of these programs by the fact that, after getting used to it, you can't believe you used to do without it. Only a few programs have made this list for me so far, and they are all staples for a programmer: a good C compiler, text editor and DOS shell. After I had used Power Windows 2.0 a few times, it too made it onto the list.

Power Windows 2.0 is a programming tool that creates Intuition data structures for you. As anyone who has programmed under Intuition knows, its object-oriented nature demands that you define structures for everything beforehand, then let Intuition deal with the structures to do what you want. The code you have to write to actually get the user input is often trivial, but setting up the data structures is tedious at best. Consider a typical program with a Screen and a Window, Boolean, String, and Proportional Gadgets (some of the gadgets using custom imagery), Menus with SubMenus, and maybe some text scattered around the window. All your program has to do is an OpenScreen, OpenWindow, SetMenuStrip and PrintIText, but the data structures could take up several hundred lines of code, each data structure containing positional and colour information, various flags, and pointers to other data structures. Setting up these structures can take hours, and every time you say "I wonder how it would look with this Gadget over here and that one two lines higher?", you have to pore over the code to find the correct values to change, then re-compile and execute your program to see the results. This is the way I used to do it, and now I really wonder how I managed to write half the stuff I did.

More Than a Time Saver

Power Windows 2.0 (PW2 from now on) lets you interactively create Screens, Windows, Requesters, all three Gadget types, Menus, and IntuiText. Each entity is added to your window(s) as you create it, so you see what you're going to get before you do a single compile. Your Window's IDCMP and other flags are set just by clicking on Gadgets – you don't have to memorize the flag names or look them up in the manual. Once created, your Gadgets may be moved, resized, deleted, or cloned. You can attach text, or any graphic imagery from an IFF file (a brush from DPaint, for example) to a Gadget or Menu item. Once in place, text or images can be moved around within a Gadget or deleted. Creating Menus is as simple as typing the text for each menu, menu item, and sub-item that you want; if, however, you want to define the position for every Menu element yourself, you can. All PW2 operations are done through simple Menu selections, and you can get by quite nicely without looking at the manual at all (which is just as well, because the manual isn't especially easy to look at). Only when your user interface is set up just the way you like do you actually get PW2 to generate the code.

Code can be generated for everything you've created in three formats: Aztec C, Lattice C, and Assembler. TDI Modula 2 is also supported, but only indirectly; you have to run a supplied program on a file produced by PW2. Inovatronics promises support for other languages in the future, and you should contact them for news of additions to the list.

The source file produced consists of initialized data structures with standard names, like "NewScreenStructure1", with optional comments explaining each element of the structures. All links between structures are set up, and #defines are made for all the structures that must be used in the executable part of the code. A source file that comes with the package uses these defines to conditionally compile statements that will produce a program to bring all the data structures to life; all you have to fill in is the routines that handle events for the user's selections. Experienced programmers will probably want to write all the code themselves and just use the data structures created by PW2, but with the source code provided, a novice C programmer with little Amiga experience could create a truly Intuitionized program.

Power Windows is an incredible timesaver, but it's more than that. Anyone who has it will probably make their programs much more interesting, fun, and easy to use, because making attractive control panels and positioning Gadgets just right becomes not only easy, but quite a blast. Having to create all the structures by hand and code images as hex values can dull any programmer's enthusiasm for exciting and colourful programs; many would just as soon put in some plain rectangular Boolean gadgets, vowing to put in the fancy features "later" (never). Not only that, but advanced Intuition programming is now open to people who have never got around to learning all the ins and outs of every data structure. In this way, PW2 improves life even for people who don't use it, by raising the general quality of Amiga programs.

One of PW2's especially interesting features is the "Grab a Window" function. This lets you point to any window in the system and instantly make a copy of all its Gadgets and Menus, the Screen it's on, etc. This way, you can run a program that you've already written and spruce it up by moving Gadgets, adding images and borders, or anything you feel the urge to do from the PW2 environment. Incidentally, you can also work on a project over several sessions in another way: instead of saving your work as source code, you can save it in PW2's own format, so that you can load it in another time to continue working with it.

PW2's user interface is completely Menu driven (big-M Menu, i.e. Intuition Menus), which might not be as convenient as Gadgets, but is much less confusing in this case since you are creating Gadgets of your own. What makes the Menus a bit less friendly than they might be is that every time you choose something, there is a pause of about a second, during which the mouse pointer displays a "HANG" message and nothing happens. In order that the Menus you're creating don't become mixed up with PW2's, the program's window pops up whenever you press the menu button, letting you make a selection from its own Menus. You can also activate the window that you're working on to see any Menus you've created yourself.

Many functions that you select from the Menu require you to take some action immediately afterwards, for example selecting, dragging or resizing a Gadget. When you make the Menu selection, a requester pops up telling you what to do next, so you have to click OK on the requester, then do what it told you to. This system is quite helpful and friendly – for the first five minutes. After that, you may curse the requesters for annoying you. Fortunately, there are several options you can set from one of PW2's menus, and turning off the helpful/annoying requesters is one of them. Even better, you can save your options so that PW2 will come up next time the way you like it. That kind of custom-tailoring in a program scores high marks with me.

Nit Picks

Is PW2 perfect? Well, I know of no competing product right now, so even if it isn't, it's a heck of a lot better than the alternative. I have no major criticisms of the product, but I can think of a few things that could make it better.

For one, a simple graphics editor should be on hand for creating simple imagery without having to bring up DPaint or another full-blown art package. It would be more convenient, and wouldn't assume that everyone has an IFF graphics program.

It would be nice if sprite data structures were supported, so that you could define a pointer; many programs at least need a "sleeping" pointer that pops up when user input is being ignored pending some operation. Alerts (messages that come up like a Guru Meditation error) aren't supported either, and while they are not often used, including them would have taken care of *every* Intuition-supplied feature.

An undo feature, while admittedly not easy to put in, could spare the user the frustration of deleting the wrong window, like the one he just spent half an hour adding Gadgets and Menus to.

Then there's that, er, manual. It's three small sheets and a cardboard cover stapled together in the middle. It has no table of contents, and is printed in tiny (albeit readable) type. If you actually force yourself to read it, however, it's helpful and not too badly written. Fortunately, the program is easy enough to use without frequent references to the manual, but it's always nice to get something substantial for your money rather than something you feel like throwing away with the plastic wrapper.

Perhaps what I miss most in PW2 is a way to define your own names for all the various structures instead of using the default names picked by the program. You can do so for Gadgets, but not for the NewScreen, NewWindow, IntuiText and other structures that PW2 creates. The structure name should be another field in the "define characteristics" window for a given object, filled in with the default name but able to accept the name of your choice. This would allow you to use existing programs or functions with newly-created PW2 code. You can always get around the problem with #defines, of course (e.g. #define newscreen NewScreenStructure), but that involves searching through the structures to find what's what, instead of thinking in terms of symbolic names from the time you create the objects.

As a final niggle, PW2 assumes that you will #include the structure definition file with your program, but I prefer to keep it as a separate module and reference the needed elements as externs – why re-compile all the structures every time when you're not changing them? It would be nice if all elements that were needed by the main program – NewWindow, NewScreen, colour palette, Menu, etc. – were declared as externs by the supplied source file, instead of having the entire file with the structures being #included. Also on the subject of the supplied C source code, the event handling function uses an IF for each condition, instead of using a SWITCH statement as it should. But all of these are minor points, and aren't of great concern in the everyday use of the program.

Power Code

So, is the addition of Power-Code to your life worth the bucks? If you program in C or assembler on the Amiga using the Intuition user interface, I would say yes. Definitely. On the other hand, you may want to become proficient at hand-coding the various structures before you spoil yourself rotten with a coding servant to attend to your every whim. Decisions, decisions.

saf-T-net

A hard disk backup utility

Review by Larry Phillips

From: R.S.N Software Inc.,
4122 S. Parker Road,
Aurora, CO 80014.
(303) 690-0102.

Price: $49.95 (US)

There are only a few unavoidable things in life, and for owners of hard disk drives, there is one more: data loss.

Loss of data is something that will happen to every hard drive owner sooner or later. Consider the many ways you can lose data on a hard drive. There is the worst case, a 'head crash', in which the surface of the disk is destroyed, necessitating a complete replacement of the unit, with consequent loss of most, if not all of the data. There is the possibility of a malicious 'trojan horse' program, or perhaps a runaway piece of code that you are working on. Unintentional power downs or AC line surges, a blown IC in the controller, misplugged cables, and even 'cockpit error', virtually guarantee that sooner or later you will lose data. Murphy's Law comes into it, of course, ensuring that it is a *lot* of data, consisting of the most important files you have. (If you've ever typed 'DELETE DIRNAME #?' instead of 'DELETE DIRNAME/#?', you'll know exactly what I mean by 'cockpit error'.)

On floppy disks, you can lose 880K of data at most. On a hard drive you can lose 10, 20, 50 megabytes or more. To the average hard drive owner,

this can mean hours of tedium and frustration trying to rebuild the disk just the way it was, searching through all those floppies for those critical programs or data files. That is, the typical hard drive owner just crosses his fingers and hopes for the best. Unfortunately, this method only works until you inevitably lose something.

The answer to the problem is simple. Backups! I can hear the moaning and groaning now. "But backups are boring!", "I don't have time!", or words to that effect. Yes, backups are boring, and do take time, and that's why I recommend saf-T-net. It is reliable, fast and, because it does not take over the machine, allows you to be doing other things while backing up your valuable data.

Saf-T-net takes advantage of the Amiga's Intuition interface, and has two main screens. The configuration screen is where you tell it the operation you want performed (backup, restore, build index). The main screen lets you start or stop the selected operation or quit the program altogether. Any device name may be used for the 'Backup Node', a term used to denote the drive you want to back up. All 4 possible floppy drives are supported (Df0:, Df1:, Df2: and Df3:).

Saf-T-net will handle both full and incremental backups, with full and selected restoration of files, selective inclusion/exclusion of directories (including wildcards). As a bonus, it also builds a handy index of all files backed up.

Full backups, as the name implies, will back up the entire disk unless you have specified that certain files and/or directories are not to be backed up. This can be done by using either an 'include' or an 'exclude' file mask. The file mask may be either a single wildcard pattern or the name of a file containing a list of filenames to include or exclude from the backup. Wildcards are also allowed if a file is specified.

The files are backed up to floppy diskettes in a non-standard format. In other words, they are not in AmigaDOS format. This has the advantage of getting the most out of each floppy in terms of data storage, and helps speed up the backup/restore process. It has only one disadvantage, and that is that the files may be read by saf-T-net only, meaning that you must have a backup of saf-T-net. Fortunately, saf-T-net is not copy protected. There are no keywords to enter or dongles to misplace, so the first order of business is to backup the backup utility.

The main screen will show you what's happening at any given time, showing the file currently being backed up, and flashing error messages when appropriate. The error messages are not left on the screen for long, but are thoughtfully written to a file in RAM: so you can see them later. There are three levels of severity indicated in the error file, ranging from *Fatal* through *Serious*, to *Recoverable*. *Fatal* errors will halt processing immediately, *Recoverable* errors may be safely ignored, and *Serious* errors should be looked at carefully. *Serious* errors imply that something went wrong that may have resulted in a file not being backed up. Processing will continue, and the file or files in question may be checked and dealt with manually, with the help of the error file.

While saf-T-net is busy, you may wish to click the screen into the background and carry on with something a little more interesting. Saf-T-net will let you know when a new floppy is needed by popping back into the foreground and putting up a requester. Each floppy takes an average of about three minutes. The exact time will depend on the number of files that are present in the area being backed up – a lot of small files will be slower than a few large ones. In saf-T-net's special storage format, each diskette holds just over 900K, so it is easy to figure out about how many you will need.

The floppies need not be formatted. If they are not, saf-T-net will do the formatting for you; and if they are, saf-T-net will write to the disks without reformatting. One feature I particularly like is that if a bad track is found on a formatted diskette, saf-T-net will format it 'on the fly'. If this fails, saf-T-net will attempt to recover by 'backtracking' to the beginning of the current floppy being written. If this operation is successful, nothing is lost. You merely put in a good disk and tell saf-T-net to continue.

Incremental backups are appended onto the set of floppy diskettes you made when the full backup was done. You only need to do a full backup when the set of backup floppies gets too large. Incremental backups only write out files that do not have the 'archive' bit set, so you are not wasting time rewriting files that are already backed up. With a logical approach to backups, very little time is needed, perhaps as little as 15 to 30 minutes per week. It's a small price to pay for peace of mind.

The index that saf-T-net builds for you is a very handy feature indeed. Sorted by filename, it becomes a way of checking to see if you have any duplicate files. Sorted by date, you will find that some of the older files might be 'expendable', and can be removed from the hard drive. It even serves as a quick way to find out which directory you put that certain file in (just so you'd know where to find it later). The larger your hard drive, the more likely you are to find a good use for the index.

Saf-T-net comes with a 30 page manual that contains a simple explanation of how to get started, a full list of all gadgets and their meanings on both screens, and a list of error messages and their meanings. The error messages are indexed alphabetically for ease of use. One page is devoted to technical assistance. The author, Bob Rakosky, can be reached via CompuServe, either through E-mail (his PPN is 75156,336), on the Amigaforum message base, or (many evenings) in the Amigaforum conference area. Support is also available through R.S.N Software during normal business hours.

You know, I always hate to read a review and find that the author hasn't mentioned any bad points about a product, but I'm afraid I can't find anything to complain about with this one. It is fast, thorough, and has a lot of built in. . . well. . . safety nets.

BYTE BOX

Amiga 500 0–2 Megabyte RAM expansion unit

Review by Chris Zamara

From: BYTE by BYTE Corporation
Arboretum Plaza II
9442 Capital of Texas Highway North
Suite 150, Austin, TX 78759

Price: $299.95 US (0k unit)
$399.95 US (512k)
$499.95 US (1 Megabyte)
$699.95 US (2 Megabytes)

Unit Tested: $299.95, 512k version

Pssst! BYTE by BYTE's got a great deal on an Amiga 500 RAM expansion unit – only 300 bucks! How much RAM? Well, this particular model comes with 0k. None. No! Wait! Come back. . .

Actually, selling a RAM expansion without memory chips may be the best solution for manufacturers in a time of wildly fluctuating chip prices. It's hard for a company to set a price for a unit full of expensive chips: they lose money for every unit sold if chip prices go up, or lose business to

competitors who can buy cheaper chips if the price goes down. It's a good deal for the user (you) as well, since you can get the bare unit to start with, then stock it up with chips when you find the best price. You can also purchase a 512k unit to get started with, then fill it up with chips as your budget allows.

Many users would rather pass up playing stock market on the price of chips and then installing them themselves, and just get one of the stocked units and plug it in. At only an extra $100 per 512k, that's still a good deal, and no matter what size you get, you can always eventually upgrade to the full two Megabytes.

No matter how much memory you do order with the BYTE BOX, what you get is an enclosed box no higher than the Amiga 500 that plugs into the expansion port and sprawls across your desk to the left of the machine. Unlike expansion units for the Amiga 1000 like Comspec's AX1000, which is tall and skinny, the BYTE BOX is wide, long and flat: 6¼" by 9¾" by 1½". Designing expansion hardware to plug directly into the A500 isn't easy, and BYTE by BYTE may have a good solution here: despite the large resulting footprint of the system, you don't lose desk space because the box itself is so wide and flat that it *becomes* part of your desk.

The unit's aluminum case with rounded top edges is easy to look at sitting next to the A500, although the colour match is not quite exact. An LED on the front panel indicates power to the unit, which turns itself on and off automatically with the computer.

The BYTE BOX comes with its own power supply to take some of the load off the A500, known for its minimal ability to supply power to external devices. The rating on our Amiga's power supply indicated it would be well up to the job of powering the system with the RAM expansion, but Scott Peterson at BYTE by BYTE explained that the Amiga circuitry has trouble supplying power to external units, even if the power supply can handle it. Whatever the technical details, it's good to see that every effort was made to prevent the BYTE BOX from cooking a user's cherished machine.

Opening the case (by removing four screws) reveals the insides to be just as pretty as the exterior. Even though the design is brand new, our test unit had no jumpers or cut traces, and appeared to be of very high quality construction. A four-layer PC board is used, which is good for low ground/VCC noise. High quality IC sockets are used for all the memory chips – a good thing, since many units will have RAM rammed into them by people who have never held a chip before. And although the unit is relatively sturdy, it is very easy to open it up to see what's under the hood. A look with a scope showed the design to have minimal wait states (where the CPU is momentarily halted): only an average of 350 nanoseconds in wait states every 14 microseconds. This is in line with the best Amiga RAM expansion units available; it is so close to zero wait states that you really are getting peak system performance.

On the negative side, the unit has no pass-through for other expansion hardware. This limitation is really a bit beyond BYTE by BYTE's control, since the 500's physical layout makes designing such expansion units difficult, especially for a reasonable cost. It remains to be seen whether other memory boards come out with pass-through, but of course if you don't plan on adding anything else (or add only units that do have pass-through), then it's not a concern.

Something that *was* of concern to us when the unit's top was off was how hot the voltage regulator's heat sink got, even though our unit was only filled with 512k worth of chips. It made us wonder if the regulator wouldn't get close to the thermal shut-down point with all two Megabytes installed, but Scott at BYTE by BYTE assured us that they have many two Megabyte units operating continuously, without a single failure. It can give you a nasty burn when poking around inside a live unit, but the manual warns to disconnect all power before opening the case for that very reason.

Installing a unit with RAM chips already in it is very simple – just plug it into the Amiga's expansion slot, plug in the power supply, and turn on the computer. The memory is automatically configured by the Amiga when you boot, so you don't have to use "AddMem" or any such boot-time utility to install it.

If you are installing your own chips, you have a bit more work to do first. The supplied manual (ten photocopied pages stapled at the top left corner) gives detailed instruction to help you, including ideas on where to buy chips, and a list of different chips that have been tested with the unit. You need 16 chips for each 512k, so as long as you can get them for less than $6.25 (US) each, you save money buying them yourself. You must upgrade from one Megabyte directly to the full complement of two Megabytes; 1.5 Megabytes is not supported, the manual explains, so that the design can follow Amiga's auto-config specifications. All that's required besides physically plugging the extra chips in is to correctly configure two plug-in jumpers on the PC board to reflect the amount of RAM installed.

Once you have installed the chips, you can test the unit completely to find out what chips are bad, if any, with some truly excellent software that comes with the package. With the unit in TEST mode (set by removing a jumper on the board), you can run a battery of tests on the RAM to thoroughly check it out in a number of ways. The program is fully Intutionized, easy to use, and contains lots of help and reference information that you can read after the tests. If the RAM does not check out, the BYTE BOX PC board is shown in a graphic display with the problematic chip or chips flashing. This makes it easy to look for bent pins, or to isolate a chip that may actually be bad.

The program is really well thought out and designed to not let anyone screw up. In fact, it didn't even let *me* screw up when I plugged in the board and ran the test without reading the manual (a bad habit that always costs me more time than reading the manual would). The program patiently explained that I had to remove the TEST jumper on the board before running the test program, and referred me to the pages in the manual that explained the procedure and showed a diagram indicating the location of the jumper! The combined factors of the BYTE BOX's design, the manual, and the test program make it possible for *anyone* to install their own chips successfully. Our test unit even came with an IC-pin straightener to persuade new chips to slide easily into place in the sockets.

Regarding the unit's performance once installed, well, a RAM board is a rather unglamorous piece of hardware in that its spectrum of performance ranges from "it works" to "it doesn't". This one does. As with any RAM expansion on the Amiga though, it increases system performance because it is "fast" RAM and eliminates contention between the CPU and the custom chips. What that means is that the machine's performance under certain conditions – such as when there is a 640 pixel wide by 16 colour screen being displayed – will improve dramatically. And some applications, like Deluxe Paint II, work faster with more RAM in the system because they don't have to keep swapping bits of the program in and out.

Although this unit is designed for the Amiga 500, it works on the 1000 as well, and in fact it is possible that some of this very text I am writing now (or you are reading now. . . hmmm, when is "now"?) is being squirrelled away in the thing – it's hanging off my workhorse A1000 that currently sports three Megabytes. It has to be turned around so that the power cord comes out the front, and it has to be supported somehow from underneath, but with an Anchor Volksmodem 12 stuffed in the gap, it works fine. It even *increased* the amount of usable desk space, because the modem was useless for putting bits of paper and disks on top of, while the BYTE BOX performs these vital tasks admirably.

So what's the bottom line on the report card? It's a well-made and well-designed box, easy to install and expand, and it's relatively cheap. In our books, that rates an A.

PLACE

STAMP

HERE

Transactor
85-10 West Wilmot Street
Richmond Hill, Ontario
L4B 1K7

News LINK

This section of *Transactor for the Amiga* is supposed to go *inside* the magazine, and henceforth it will. It's the place where we keep you up to date on prices and policies for our products, and where we put our Industry News (i.e. press releases). This time we just plain ran out of room, so we had to put it here, cut down our own announcements to a minimum, and eliminate the Industry News altogether.

Uedit

Those of you who have been living on this planet the last year or two have probably heard of Rick Stiles' amazingly flexible Uedit text editor. To you others we'll just say that we use Uedit ourselves because it's fast, friendly and fully configurable. You can have up to 100 buffers for text, and switch instantly from one to another – great for programs with multiple modules. A split window facility has now been added to the brand-new version 2.3, plus a whole lot of other features we don't have room to mention.

Because we like Uedit so much, we think it deserves a wider user base than it currently has, so we've arranged with Rick Stiles to make it available through our magazine. If you're a programmer and haven't tried Uedit yet, we think you should give it a try. The price? Just $45.00 (US) or $58.00 (Cdn), plus the usual couple of bucks for shipping and handling.

Fish Disks

We like to sell stuff we like, so we had a chat with Fred Fish and asked him if he'd mind us distributing his famous – and rapidly growing – library of public domain and shareware programs. That was fine with Fred, especially when we told him we'd spruce up the disks with our own typeset labels. Big deal, you say? Well, these aren't ordinary labels – the entire contents of each disk is given on the label along with, in most cases, capsule descriptions. No more taking endless directories of your Fish disks to find the program you're looking for. . . just flip through your collection and you'll find the file name in easy-to-read type right on the label. Our price: $7.00 (US), $8.00 (Cdn).

Transactor for the Amiga Disk for Volume 1, Issue 1

One thing you'll notice about our magazine is that we print a lot of source code. This is mostly for reference and study purposes – we don't really expect you to type it all in (though you're welcome to if that turns you on!). That's why we offer a disk subscription in addition to the magazine subscription. There's one disk per magazine, containing the source, executable and documentation for every program in that issue, sometimes in more elaborate versions than we have room for in the magazine itself, and often with supporting files in the "nice but not totally necessary" category. If there's room on a particular disk we also include any neat PD or shareware stuff that's caught our eye lately, again with documentation and, where possible, source code.

. . .continued on flip side

Reader Service Card

Fill out the card (please print) or affix your peel-off mailing label. This card valid until June 30, 1988.

Name _____

Company _____

Address _____

_____

City _____ *Prov/State* _____

Postal/Zip Code _____

Phone# _____

See Advertiser Index
for corresponding service numbers

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130
131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170
171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190
191	192	193	194	195	196	197	198	199	200

We haven't actually put together the first disk in the series as we go to press, but it will be ready by the time the magazine gets back from the printer. Here's some of the stuff we expect to include: *all* the ARP materials, including the replacement AmigaDOS commands plus docs, arp.library plus docs, the C bindings for the library for both Manx and Lattice, and all the necessary include files; the Warrior Cycles program with the complete set of players written so far; Rob Peck's mini list demo, Matt Dillon's DOS Packet demos and the other programs in the magazine; and, if space permits, a totally revamped version of our own well-known Structure Browser program. For ordering info, see the attached order card.

Amiga Classic Disks 1 and 2

This Amiga-specific magazine is a new departure for us, but we've actually been covering the Amiga for quite a while in the pages of our original magazine, *Transactor*. Rather than let all the good programs we published fade into obscurity, we've put them together on a special two-disk collection: Transactor Classic Amiga Disk #1 (which has already been out for a while), and Transactor Classic Amiga Disk #2 (this one's brand new). We don't have space to list their contents here (maybe next time), but they're both filled just about to overflowing with the best programs we could find, from our own pages and from the FRD (Freely-Redistributable Domain) – a new term we invented right now and humbly offer to the world so that no one will ever again have to wrap their lips around the word "redistributable".

Contest Winners

At the World of Commodore show last December, a survey was taken by The Amiga Developers Forum based here in Toronto. Twenty subscriptions were offered as prizes to those who completed the survey forms. For the winners who paid for a subscription (most did), we'll simple extend those subs by another year. Thanks to all who took the time to return survey info and perhaps we'll have the results by next issue. The winners are: Viki Kapoor, Bramalea, Ont.; Geoff Park, Toronto, Ont.; Richard G. Wolfe, Brampton, Ont.; Ronald L. Headley, Painesville, Ont.; Mark Kwan, Maple, Ont.; Elizabeth Arnold, Richmond Hill, Ont.; Chris Davies, Orangeville, Ont.; Paul Howard, Peterborough, Ont.; D.R. Moxham, Burlington, Ont.; Jim Doyle, Mississauga, Ont.; Steve Hill, Toronto, Ont.; Allan Persawd, Don Mills, Ont.; David Lesle, South Euclid, Ohio; Henry Grayson, Mississauga, Ont.; David Evans, Orton, Ont.; Bill Grixti, Gravenhurst, Ont.; Paul Cowan, Brampton, Ont.; Al Bates, Washago, Ont.; , Wilson Chan, Scarborough, Ont.; and Tom Eyre, Mississauga, Ont.

Advertiser Index

Advertiser	Page #	Card #
Datamax Research Corp.	1	11
The Hunter Group	OBC	20
MCP Associates	IFC	45
ReadySoft Inc.	1	61
R.S.N. Software, Inc.	1	80

(* Prefer to be contacted directly)

for the AMIGA Transactor
The Magazine for Amiga Programmers

☐ Please send me **6** consecutive **Transactor for the Amiga** starting with the next issue!
☐ I'm subscribing to the magazine AND the disk, so please send me my **FREE Fish Disk Catalog Disk.**
☐ I am already subscribing to Transactor – my subscription # is _____ :
☐ Please switch my Transactor subscription to Transactor for the Amiga (no charge).

Subscriptions (☐ Please send microfiche instead of magazines)

	Canada ($Cdn.)			U.S.A. ($US)			Foreign ($US)			Air Mail ($US) (Overseas only)		
	Mag	**Disk***	**Both***	**Mag**	**Disk**	**Both**	**Mag**	**Disk**	**Both**	**Mag**	**Disk**	**Both**
1 Year	☐19.00	☐67.00	☐86.00	☐15.00	☐55.00	☐70.00	☐21.00	☐65.00	☐86.00	☐40.00	☐79.00	☐119.00
2 Years	☐35.00	☐123.00	☐158.00	☐27.50	☐101.00	☐128.50	☐38.50	☐119.00	☐157.50	☐73.00	☐145.00	☐218.00
3 Years	☐47.50	☐167.50	☐215.00	☐37.50	☐137.50	☐175.00	☐52.50	☐162.50	☐215.00	☐100.00	☐197.50	☐297.50

* Ontario Residents please add 7% of disk subcription price for provincial sales tax. No PST on magazine subscriptions.

☐ New Subscription ☐ Renewal (please include your Subscription Number from mailing label) _____

Name & Address ____ New address? ☐ _____
(please include your postal/zip code):

SAVE 28% OFF Cover Price
(one year sub)

Phone #: ( ) CompuServe ID#:

*** Ontario residents please add 7% PST † Prices are in U.S. Dollars**

Add flat $2.00 shipping & handling for any number of items within this box

$Cdn.	$U.S.	
☐ 15.95*	13.95†	Transactor T's ☐S, ☐M, ☐L, ☐XL ☐Red ☐Blue
☐ 19.95*	17.95†	Jumbo T-Shirt ☐Red ☐Blue

$Cdn.	$U.S.	
☐ 14.95*	12.95†	Transactor Classic Amiga Disk #1
☐ 14.95*	12.95†	Transactor Classic Amiga Disk #2
☐ 58.00*	45.00†	Uedit Version 2.4 (2 disks)

Note: See News LINK for details on above items

Transactor for the *AMIGA* Disk $12.95† US, $14.95* Cdn each.
(correspond to issues of Transactor for the *AMIGA*)
☐ Vol 1, Issue 1

Fish Disks $6.95† US, $7.95* Cdn each.
Note: Disks numbers not yet released may be ordered in advance – simply add numbers to end of list. Disks 57, 80, and 88 not available.

☐ 1	☐ 11	☐ 21	☐ 31	☐ 41	☐ 51	☐ 62	☐ 72	☐ 83	☐ 94	☐ 104	☐ 114	☐ 124
☐ 2	☐ 12	☐ 22	☐ 32	☐ 42	☐ 52	☐ 63	☐ 73	☐ 84	☐ 95	☐ 105	☐ 115	☐ 125
☐ 3	☐ 13	☐ 23	☐ 33	☐ 43	☐ 53	☐ 64	☐ 74	☐ 85	☐ 96	☐ 106	☐ 116	☐ 126
☐ 4	☐ 14	☐ 24	☐ 34	☐ 44	☐ 54	☐ 65	☐ 75	☐ 86	☐ 97	☐ 107	☐ 117	☐ 127
☐ 5	☐ 15	☐ 25	☐ 35	☐ 45	☐ 55	☐ 66	☐ 76	☐ 87	☐ 98	☐ 108	☐ 118	☐ 128
☐ 6	☐ 16	☐ 26	☐ 36	☐ 46	☐ 56	☐ 67	☐ 77	☐ 89	☐ 99	☐ 109	☐ 119	
☐ 7	☐ 17	☐ 27	☐ 37	☐ 47	☐ 58	☐ 68	☐ 78	☐ 90	☐ 100	☐ 110	☐ 120	
☐ 8	☐ 18	☐ 28	☐ 38	☐ 48	☐ 59	☐ 69	☐ 79	☐ 91	☐ 101	☐ 111	☐ 121	
☐ 9	☐ 19	☐ 29	☐ 39	☐ 49	☐ 60	☐ 70	☐ 81	☐ 92	☐ 102	☐ 112	☐ 122	
☐ 10	☐ 20	☐ 30	☐ 40	☐ 50	☐ 61	☐ 71	☐ 82	☐ 93	☐ 103	☐ 113	☐ 123	

Add flat $2.00 shipping & handling for any number of items within this box

Transactor for the *AMIGA* Back Issues and Microfiche: $4.50 US, $4.50*Cdn. (MF = Microfiche)
☐ Vol 1, Issue 1 ☐ MF

Please allow 6-8 weeks for delivery in U.S., 4-6 weeks in Canada
NOTE: Prepayment required. Purchase orders will be accepted ONLY if accompanied by payment.

☐ Visa ☐ MasterCard ☐ Cheque/MO. enclosed Cheque# _____ Dated ____/____/____ Amount _____

Acct. # _____ Expires ____/____ Signature _____

☐ Please send advertising information for Transactor. ☐ Please send dealer information for Transactor. 12/87

fold along scores with correct reply card on outside – use tape, no staples or glue please

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 390 BUFFALO, NY

POSTAGE WILL BE PAID BY ADDRESSEE

The Transactor

P.O. Box 338 Station 'C'
Buffalo, NY, 14209-9990

Business Reply Mail
No Postage Stamp Necessary if mailed in Canada

Postage will be paid by:

Canada Post Postes Canada
46487

The Transactor

501 Alden Road
P.O. Box 3250
Markham Industrial Park
Markham, Ontario
L3R 9Z9

THE TIME SAVER

Type in a lot of Transactor programs?
Does the above time and appearance of the sky look familiar?
The Transactor Disk has saved countless thousands of hours for would-be typists worldwide!
Transactor for the *AMIGA* 3½" Disks will add to the thousands, especially with the custom labels that will list each directory right on the disk!

Single 3½" Disk Orders Only $12.95 US, $14.95 Cdn. Per Issue
6 Disk Subscription (one year) Just $55.00 US, $67.00 Cdn.
(see order form at center fold)

And don't forget the Classics from the past!

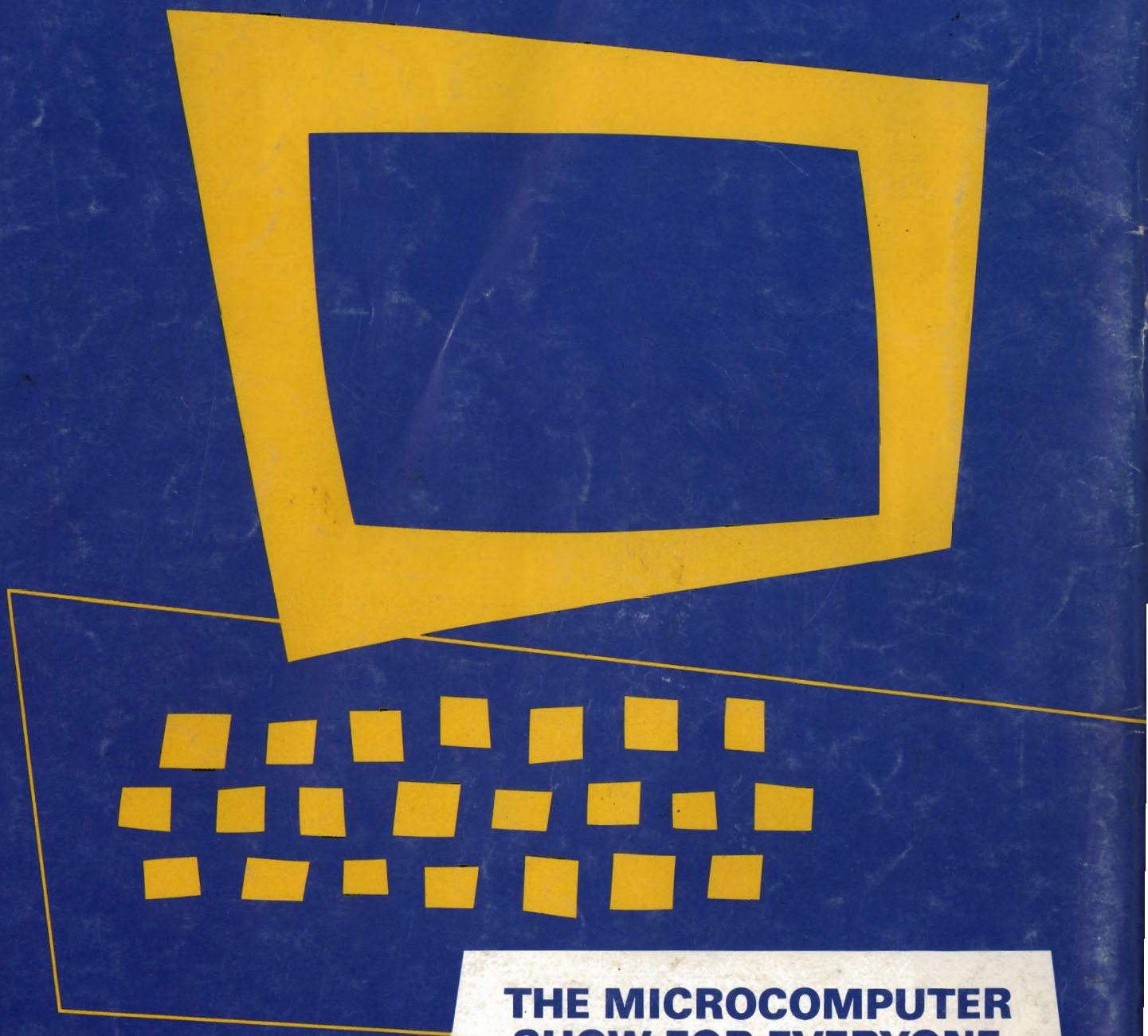
Transactor Classic Amiga Disk #1 and #2, $12.95 US, $14.95 Cdn.

All the Amiga programs from Transactor Classic magazine Vol6, Issue 6 to Vol8, Issue 1, and Vol8, Issue 2 to Vol8, Issue 5. Both with complete documentation on disk, plus our pick of the public domain!

Computer Expo

April 14, 15, 16, 17, 1988

Toronto International Centre



THE MICROCOMPUTER SHOW FOR EVERYONE

Produced by The Hunter Group Inc. (416)595-5906